ADAPTIVE CPU-BUDGET ALLOCATION FOR SOFT-REAL-TIME APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Safayet N. Ahmed

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2014

ADAPTIVE CPU-BUDGET ALLOCATION FOR SOFT-REAL-TIME APPLICATIONS

Approved by:

Professor Bonnie H. Ferri,
Committee Chair
School of ECE
Georgia Institute of Technology

Professor Magnus Egerstedt
School of ECE
Georgia Institute of Technology

Professor Yorai Wardi
School of ECE
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
School of ECE
Georgia Institute of Technology

Professor Karsten Schwan
School of CS
Georgia Institute of Technology

Date Approved: 28 April 2014

To my family,

my parents Nizam Ahmed and Mariam Sultana

and my brother Rubayet Ahmed

ACKNOWLEDGEMENTS

I would like to start by thanking Professor Bonnie Ferri, who consented to be my PhD advisor and provided much needed help and guidance throughout my PhD career. More importantly, professor Ferri was patient with me at times when I may have been difficult and gave me time to mature and grow as a researcher.

Additionally, I would like to thank the members of my reading cmmittee, Professor Sudhakar Yalamanchili and Professor Yorai Wardi for their help and encouragement.

I would like to thank Dr. Muhammad Muqarrab Bashir, Dr. Arif Selcuk Uluaguc, Dr. Aleem Mushtaque, Dr. Jeffrey Young, and Dr. Shahnewaz Siddique as well for their help, guidance, and encouragement. I would like to thank Michael Giardino for his help implementing significant aspects of LAMbS. Also, I would like to thank my friends and peers, Musheer Ahmed, Talha Khan, Robert Palmer, and Dr. Daniel Lertpratchiya for making the PhD process a bit more bearable. It always helps to know people who are in the same boat. In addition, I would like to thank Muhammad Rizwan, Elhadji Alpha A Bah, Usman Ali, Aftab Ahmed Khattak, and Hamza Abbasi for their support.

I would like to thank my parents Nizam Ahmed and Mariam Sultana for their prayers, patience, and sacrifice throughout my years as a PhD Student. I could not have come this far without their help and support. Lastly, I thank Allah, the Most Mericful and Wise, for all His blessings and for keeping me in the company of good people.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

From smart phones and multi-media players to mobile gaming platforms, mobile embedded devices are ubiquitous. Given the demand for such devices to have higher performance, longer battery life, and a lean form factor, power management is an essential component of such devices [36, 54]. Power management has also become a key issue for servers and data center operations [16]. The total cost of energy for data centers includes not only server electricity bills, but also the cost of energy for the cooling infrastructure and provisioning costs. Power-management is essential to manage the cost of operating such large-scale systems.

In both mobile embedded devices and large-scale servers, the performance demand for the computing system is not uniform over time. The peak demand for computing performance is often significantly higher than the average-case demand. "Mobile use-case studies show that most mobile devices are typically in active standby state for eighty percent of the time, and process intensive mobile applications twenty percent of the time" [3]. Similarly, most of the time, "servers operate between 10 and 15 percent of their maximum utilization levels" [16]. Such variability in load motivates the use of platforms with modes of operation where the different modes of operation offer different levels of trade-off between low power and high performance.

Typically, the modes of operation supported by servers or mobile embedded platforms are based on dynamic-voltage-scaling (DVS), clock gating, and power gating. Recent mobile embedded platforms support more exotic power-management mechanisms based on heterogeneous multiprocessors. All of these power-management mechanisms require a software component to monitor the workload, to determine

the appropriate level of trade-off between low power and high performance, and to choose the mode or modes of operation that will result in the desired level of trade-off.

The appropriate level of trade-off depends on the goals of the system. In many large-scale data centers, there may be limits on the maximum power consumption supported by the infrastructure. In such cases, the goal may be to maximize performance given maximum power constraints. Also, there are cases where servers host time-sensitive applications, and the goal is to minimize power consumption given timing constraints or performance constraints. In this dissertation, the latter case is addressed. Specifically, the focus of this dissertation and the presented system is power-management under soft timing constraints.

There is a significant body of previous work to address the problem of power management of systems hosting real-time applications, and this work is primarily based on dynamic voltage scaling. However, much of this previous work is impractical, based on assumptions that are increasingly unrealistic for modern platforms and applications, and based on a priori knowledge that is not possible to obtain without extensive off-line tuning. To address some of these issues, a novel approach is presented in this dissertation called Linear Adaptive Models based System (LAMbS).

LAMbS is based on a generic model that describes the hardware platform in terms of discrete modes of operation. No assumption is made about the relationship between power and performance associated with the modes of operation. Therefore, LAMbS is applicable to platforms where typical assumptions about DVS do not apply or even to platforms where power management is not based on DVS. Furthermore, power-management decisions in LAMbS are based on real-time measurements of the workload and real-time measurements of the power and performance associated with each mode of operation. As a result, LAMbS does not require static off-line tuning or access to any a priori knowledge. An overview of LAMbS is presented in the following section.

1.1 LAMbS: Linear-Adaptive-Models-based System

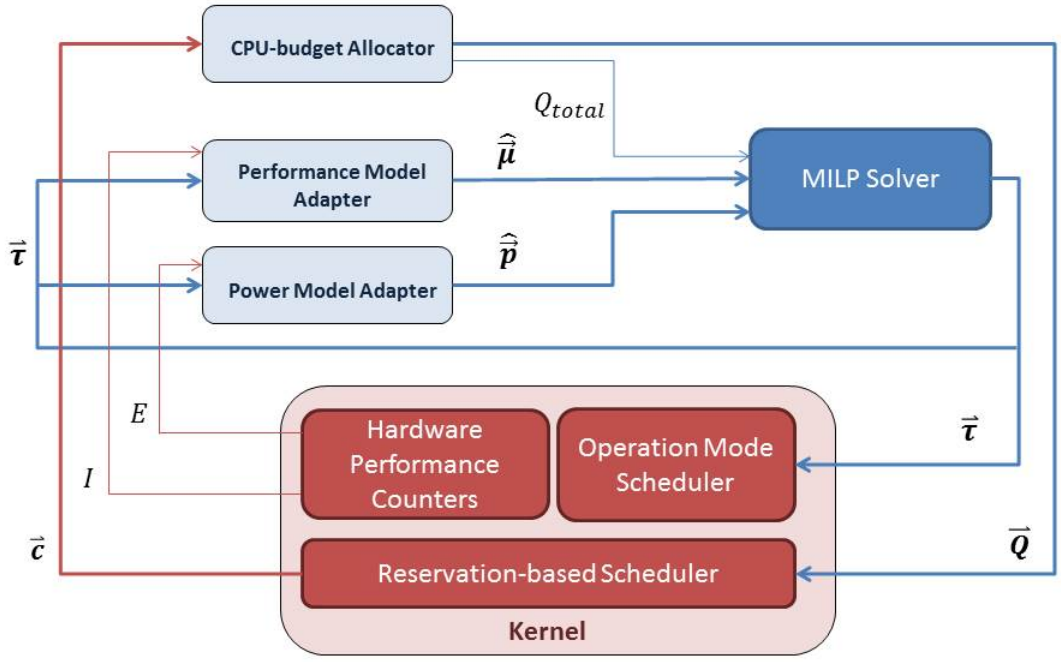


Figure 1: The overall structure of the Linear Adaptive Models-based System.

The overall structure of LAMbS is shown in Figure 1. LAMbS consists of several components: a CPU-budget allocator, power and performance model adapters, an MILP solver, and other kernel-level components. The CPU-budget allocator, model adapters, and solver make up the controller. The kernel-level components encapsulate the plant, that is, the computing system to be controlled.

In LAMbS, time is divided into periodic intervals called reservation periods. CPU-budget allocations and power-management decisions are made at reservation period boundaries. As described earlier in this chapter, it is assumed that the hardware platform is configurable to one of a number of discrete modes of operation. Each mode of operation offers a different level of trade off between high performance and low power. Power management decisions are in the form of vectors called operation-mode schedules, $\vec{\tau}$, that denote the amount of time to be spent in each mode of

operation during the following reservation period.

The inputs to the controller include the CPU-budget consumed by each task, $\vec{c}$, the total energy consumed by the system, E, and the total number of instructions retired across all tasks, I. The inputs are obtained from measurements in the plant at the end of the previous reservation period. The outputs from the controller consist of the budget allocated to each task, $\vec{Q}$, and the operation-mode schedule, $\vec{\tau}$. The controller outputs affect the plant during the following reservation period.

At the beginning of each reservation period, the scheduler computes the CPU budget to allocate to each task over the next reservation period. Typically, in previous work, CPU budget is allocated in terms of CPU time or clock cycles. However, the number of nanoseconds or cycles required to complete a job depends on the mode of operation in which the job is run. On other other hand, power-management decisions depend on the total CPU budget committed to tasks. To simplify the problem of budget allocation and power-management in LAMbS, computation is measured and allocated in terms of virtual instructions.

Virtual instruction count (VIC) is an abstract measure of computation that approximates retired-instruction count. Virtual instructions can be allocated to tasks and consumed by tasks in the same way that CPU time or clock cycles are allocated and consumed. The motivation for using VIC-based CPU budgets is described in greater detail in Chapter 5. The vector of VIC values allocated to each task is denoted $\vec{\mathbf{Q}}$ in Figure 1. The total requested VIC, denoted Q_{total} in Figure 1, is the sum of the number of virtual instructions allocated to all tasks.

Virtual instruction count is based on approximating retired-instruction count and requires an estimate of the average instruction-retirement rate in each mode of operation. Also, computing the optimal operation-mode schedule requires an estimate of the average power consumption rate in each mode of operation. As described earlier, the total number of instructions retired, I, and the total energy consumed over the

previous reservation period, E, are measured at the end of each reservation period and fed back into the controller. The vector of times spent in each mode of operation, $\vec{\tau}$, is fed back into the controller as well. These measurements are used by the model adapters to estimate the average power consumption rate, $\hat{\vec{p}}$, and instruction retirement rate, $\hat{\vec{\mu}}$, associated with each mode of operation.

The variables, Q_{total}, $\hat{\vec{\mu}}$, and $\hat{\vec{p}}$, are used to compute the optimum operation-mode schedule, $\vec{\tau}$. The optimum operation-mode schedule is one that minimizes total power consumption while ensuring that the increase in VIC over the following reservation period is greater than or equal to Q_{total}. To ensure that a solution exists for this optimization problem, Q_{total} is saturated based on the length of the reservation period and the maximum estimated instruction-retirement rate of the available modes of operation.

The kernel-level components execute the decisions of the controller. CPU-budget allocations are enforced using a reservation-based scheduler. The operation-mode schedule is executed by the operation-mode scheduler. Lastly, instruction-count and energy measurements, the timing of the reservation periods, and invocation of controller components at reservation-period boundaries are all handled by kernel-level components.

In the implementation-related details presented in this dissertation, the modes of operation are limited to CPU frequency scaling. Also, the scope of this dissertation is limited to general budget-allocation algorithms and the VIC-based budget-allocation algorithm. While possible implementation approaches are described in Chapter 6, implementation details and experimental results on the operation-mode scheduler and linear optimizer are beyond the scope of this dissertation.

1.2 Main Contributions

Overall, LAMbS is a novel approach to power management that not only adapts to the changing computational load of hosted applications but also to the changing power and performance characteristics of the managed hardware. The research presented in this dissertation makes a number of contributions.

1. First, **asynchronous budget allocation** is presented. In previous papers on adaptive budget allocation, CPU budget is adapted on job completion. In the event of a severe under allocation, there is significant delay before the job completes and the system is able to correct for the under allocation. In this dissertation, an algorithm is presented where budget allocations are adapted at reservation-period boundaries. Simulation results are presented to demonstrate the advantages of this faster rate of adaptation. Furthermore, a proof of queue stability is presented. Experimental results are presented for a linux-based implementation that demonstrates the feasibility of this algorithm and its stability under overload conditions.

2. The second contribution of this dissertation is the **two-stage prediction (TSP)** algorithm. The TSP algorithm is developed as an improvement on asynchronous-budget allocation. In this algorithm, the first stage predicts the execution time of the next job each time a job completes. Also, the first stage provides estimates of the prediction error satistics. The second stage uses the output of the first stage and additional feedback from the OS kernel to compute the appropriate budget allocation at reservation-period boundaries. Because budget allocations are updated at reservation-period boundaries, it is possible to respond quickly to under allocations.

   Also, this architecture allows each application to exploit application-specific domain knowledge in the first-stage predictor. Furthermore, the second stage of

the TSP algorithm is designed to limit the probability of jobs missing deadlines. This probability bound is proved using a variant of Chebyshev's inequality.

3. As a part of the research on TSP, a novel algorithm is presented for predicting execution times, called the **LMS-MA hybrid predictor**. This algorithm consists of an array of LMS filters and moving-average filters and is designed to exploit correlation in job-execution times. Use of LMS filters for predicting execution times has not been seen in the reviewed literature. For certain workloads, LMS filters provide more accurate predictions than simple moving averages.

   Furthermore, this algorithm is designed with the goal that it can be used without tuning. To that end, the LMS filter is used to eliminate the need to tune the weights of the filter. To eliminate the need to tune step sizes, variable step sizes are used. An array of filters of different lengths are used to eliminate the need to tune filter lengths. Also, the array includes uniformly-weighted moving averages for workloads where execution times are not correlated. Such an approach has not been seen in the reviewed literature on LMS algorithms.

4. Another contribution of this dissertation is **virtual instruction count (VIC)**. As described in Section 1.1, VIC is a novel abstract measure of computation for budget allocation. In previous work on reservation-based scheduling and power management, CPU budget is specified in terms of clock cycles and CPU time. VIC is an approximation of retired-instruction count based on estimated retirement rates. Experimental results are presented to demonstrate that VIC-based budget allocation is more efficient than time-based budget allocation in the presence of frequency scaling.

5. Another contribution of this dissertation, related to VIC, is the **adaptive**

performance model. The performance of each mode of operation is specified in terms of instruction-retirement rate. Instruction-retirement rate is a more accurate measure of performance than clock frequency. Also, intruction-retirement rate is a more flexible measure of computation that can be applied to power-management mechanisms other than frequency scaling. The average instruction-retirement rate is estimated for each mode of operation using an LMS filter and instruction counts. The estimated retirement rates are used for VIC-based budget allocation as well as for power-management decisions.

6. An **adaptive power model** is presented as well. The power-consumption rate is dynamically estimated for each mode of operation using run-time energy measurements. These rates are estimated using LMS filters in the same way that instruction-retirement rates are estimated. Accurate estimates of power-consumption rates should allow for more efficient power management.

   Efficient run-time energy measurements are possible due to a built-in energy-measurement mechanism in Intel processors that was not available until recently. It is possible to obtain accurate dynamic estimates of power-consumption rates due to such run-time energy measurements. In previous work on power management, energy was measured using oscilloscopes, multimeters, and power meters. Direct run-time measurements of energy usage was not possible. In some papers, energy usage is estimated using linear models based on architecture events. However, such models are architecture specific and have to be derived off line. In contrast, the approach presented in this paper is simpler and does not require any off-line tuning.

7. Finally, the problem of **power management** is posed differently in this dissertation compared to previous work. In most previous work, the power-management decision involves choosing the mode of operation based on the progress of the

total workload. In LAMbS, the amount of CPU capacity to be committed across all tasks is decided at the start of each reservation period. Then, the power-management decision involves choosing how much time to spend in each mode of operation over that reservation period, accommodating the committed CPU capacity. This power-management decision is based on the dynamic power model and performance model and takes into account both active modes of operation and idle states. It is shown that in this form, power-management is a mixed-integer-linear programming problem.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows. Background material and previous work on power management and scheduling are presented in Chapter 2. Power-management mechanisms are discussed including those available on commercial processors. Scheduling and voltage-scaling algorithms are discussed for hard-real-time and soft-real-time systems. Techniques are discussed for addressing non-ideal behavior on real-world platforms. The chapter concludes with a discussion on how LAMbS improves on previous approaches to power-management.

An algorithm for asynchronous budget allocation is discussed in Chapter 3. In this chapter, the real-time-task model and CPU-reservation model are discussed. The PBS algorithm is presented in detail with proof of queue stability. A Linux-based implementation of the algorithm is described. Experimental results are presented for synthetic and video-decoding workloads.

In Chapter 4, the two-stage-prediction (TSP) algorithm is presented. A first-stage prediction algorithm, called the LMS-MA hybrid predictor, is presented. The second stage of the TSP algorithm is developed based on Chebyshev's inequality. Experimental results are presented for a Linux-based implementation of the algorithm and a wide range of multimedia workloads.

In Chapter 5, virtual instruction count (VIC) is developed as an alternative measure of computation for budget allocation. Results are presented from frequency-scaling experiments to demonstrate the need for such an alternative. Examples and implementation details are presented for VIC-based budget allocation. Experimental results are presented to demonstrate the advnatages of VIC-based budget allocation over time-based budget allocation.

Future work, mainly addressing power management, is presented in Chapter 6. An adaptive power-usage model is presented. The power-usage model is modified to account for transition overheads. Based on this power model, power management is posed as a mixed-integer-linear programming problem. In addition to power management, issues arising from nonuniform workload mixes are discussed as well. Finally, a summary of this dissertation is presented in Chapter 7.

CHAPTER II

POWER MANAGEMENT OF REAL-TIME SYSTEMS

A real-time system consists of a set of tasks with timing constraints. These tasks consist of periodic bursts of computation that must be completed within corresponding deadlines. A schedule or scheduling algorithm describes how a set of tasks share the CPU over time. A task set is said to be feasible if there exists a schedule that will allow computation from all tasks to be completed on time. The feasibility of a task set depends on the rate and magnitude of the requests for CPU time and on the algorithm used to distribute time on the CPU between the tasks. *Real-time scheduling entails determining if a real-time task set is feasible and computing a schedule that allows for a timely allocation of CPU time to the contending tasks.*

Real-world tasks often exhibit variability in the use of CPU time. The average-case demand for CPU time may be significantly lower than the peak demand. On the other hand, the computing platform hosting the tasks may have modes of operation that trade-off higher performance for lower power consumption. In the common case when the load on the CPU is small, it may be acceptable to switch the platform to a mode of operation with lower performance given that this switch may lead to overall energy savings. *Power management of real-time systems entails trading-off just enough platform performance for lower power consumption such that overall energy consumption is reduced while the timing constraints of all tasks are still satisfied.*

The remainder of this chapter contains a literature review of previous scheduling and power management algorithms for real-time systems. Power dissipation and power-management mechanisms are discussed in Section 2.1. The periodic real-time task model and two fundamental scheduling algorithms are presented in Section 2.2.

Dynamic voltage and frequency scaling (DVFS) algorithms are discussed in Sections 2.3. Soft-real-time tasks and the power management of systems with such tasks are discussed in Sections 2.4. Non-ideal behavior of real-world platforms and previous approaches to managing such platforms are discussed in Sections 2.5. Finally, a comparison is presented in Section 2.6 between LAMbS and approaches to real-time power management presented in previous work.

2.1 Power Management Mechanisms

The most power consuming component of a computer system is often the processor, and processors are built on CMOS technology. Other components such as GPUs and SRAM-based caches are also built on CMOS technology. This section presents a model for power dissipation in CMOS devices and techniques used in such devices to dynamically trade-off performance for lower power consumption.

Power dissipation in digital CMOS circuits can be expressed as shown in (1) [22]. This expression consists of three different terms: switching power, short-circuit power, and leakage power.

$$P_{total} = P_{switching} + P_{short\,circuit} + P_{leakage} \tag{1}$$

$$P_{switching} = p_t(C_L {V_{dd}}^2 f_{clk}) \qquad P_{short\,circuit} = I_{SC} V_{dd} \qquad P_{leakage} = I_{leakage} V_{dd}$$

Dynamic power dissipation is made up of switching power and short-circuit power. Switching power, $P_{switching}$, is associated with charging and discharging the loading capacitance, C_L, at the output of gates. The supply voltage is denoted as V_{dd}. The probability that a power consuming transition occurs at the output of a gate is denoted p_t. The term, $(p_t C_L)$, is sometimes replaced with the effective capacitance, $C_{effective}$. Short-circuit power, $P_{short\,circuit}$, is the additional power dissipated in the short time interval during switching when both the PMOS and NMOS transistors are simultaneously active. This creates a short circuit between V_{dd} and ground and produces a short-circuit current, I_{SC}, and causes power loss.

Leakage power is a part of static power. It is associated with leakage current, $I_{leakage}$, that flows through transistors even when they are "turned off". In the past, switching power was the most significant component of total power dissipation. However, with shrinking process technology and decreasing supply voltage, the leakage component of total power becomes significant [45]. Further details of the sources of static power are presented in [45].

Most papers on CPU power management relate to systems based on Dynamic Voltage Scaling (DVS). DVS entails scaling the supply voltage and clock frequency of a device to trade off performance for lower dynamic switching power. Switching power, $P_{switching}$, is proportional to V_{dd} squared. Therefore, any decrease in V_{dd} yields a quadratic decrease in $P_{switching}$. However, decreasing V_{dd} increases circuit delay and limits the maximum clock frequency, f_{clk}. Furthermore, $P_{switching}$ is also proportional to clock frequency. As a result, a linear decrease in supply voltage and clock frequency yields a cubic decrease in switching power at the cost of a linear decrease in performance.

DVS-based power management involves reducing or eliminating CPU idle time by slowing and lengthening execution times. This is ideal if the CPU consumes power at roughly the same rate when idle as when performing valid computation. An alternative to the DVS-based strategy of shortening CPU idle time is to maintain the idle time, and then transition the CPU to one of a number of lower power *sleep states* when the CPU is idle. Sleep states can be implemented through *clock gating* or *power gating*. Clock gating may involve disabling the clock signal to components, disabling parts of the clock distribution network, or disabling clock generation all together[19]. Power gating involves shutting off the power supply to components being put to sleep.

Both DVS and clock gating only reduce dynamic power, whereas power gating also reduces static power. As in the case of DVS, there is a trade-off between power and performance in using sleep states. When transitioning to a sleep state, a system

incurs overheads in power and performance. Deeper lower-power sleep states have higher transition overheads. Transitioning to a sleep state is only possible if the CPU idle time is longer than the transition times to and from that sleep state. Also, transitioning to a sleep state is only worthwhile if the CPU idle time is longer than the *break-even time*. Break-even time is the minimum time a component must spend in the sleep state to make up for any excess energy dissipated in the sleep-state transition. Break-even time is discussed in greater detail in [19].

2.1.1 Power Management Mechanisms in Commercial Processors

Most consumer desktops and laptops use x64 processors from Intel and AMD, which implement frequency scaling. Frequency scaling is referred to as "SpeedStep" on Intel processors and "PowerNow!" or "Cool'n'Quiet" on AMD processors [41, 10]. Most Intel and AMD processors also implement one or more low-power idle states called C-states [41]. These low power states may involve different levels of clock-gating and voltage reduction or power gating [68].

The e500mc PowerPC architecture from Freescale also supports a number of idle states based on clock gating[34]. In addition to clock gating at the CPU level, the e500mc also shuts off the clock signal for some components when those components are idle.

Both power gating and DVS have been adopted in mobile embedded processors as well, such as the Snapdragon S4 system that uses the Krait micro-architecture from Qualcomm [2]. The Krait micro-architecture is described as an asynchronous Symmetrical Multi-Processor system (aSMP), referring to the fact that each core on the Krait can be configured independently to a different voltage and frequency or power gated.

An alternative approach to power management taken by ARM is big.LITTLE processing[36]. A big.LITTLE system consists of multiple processors that are identical

in the interface exposed to the system and application programmer but different in terms of the micro-architecture. The first implementation of big.LITTLE consists of a big Cortex-A15 and a LITTLE Cortex-A7. The larger Cortex-A15 trades off energy efficiency for high performance, where as the Cortex-A7 trades off high performance for energy efficiency. A key innovation in this approach is the task-migration use model enabled by the less-than-20μs task-migration time between the processors. An application only operates on the Cortex-A15 or the Cortex-A7 but not both at the same time. From the programmer's perspective, it appears that a single processor is switching the mode of operation between a low-power Cortex-A7 mode and a high-performance Cortex-A15 mode.

A similar approach called variable Symmetric Multi Processing (vSMP) is used in NVIDIA's Tegra 3 (Kal-El), [3]. The vSMP system consists of five cores that share the same ARM Cortex-A9 architecture and each core can be power-gated independently. However, one of the five cores, referred to as a companion core, is built using a special low-power silicon process that allows this core to operate efficiently at lower frequencies. Either the companion core or one or more of the main cores are active at any given time. The switching time to change from the companion core to the main cores is estimated to be in the order of $2ms$. While the number of high-performance cores that are active might change with workload, all active high-performance cores operate at the same frequency unlike the cores in Qualcomm's aSMP approach.

Hybrid Symmetric Multi-Processing (Hybrid-SMP) from Marvell uses a combination of the aSMP and vSMP approaches[54]. The Hybrid-SMP system consists of two high-performance cores, termed HPM, and a low power core, termed LPM. As in the case of big.LITTLE and vSMP, the processors are 100 percent compatible from the programmer's perspective and offer a trade-off between low power and high performance. However, as in the case of aSMP, all processors may be operated

simultaneously and voltage scaled and power gated independently. In addition, components associated with unused features of the processor like SIMD operations are clock gated.

2.1.2 Power Management of Additional Components

In the above discussion, only CPU power management is addressed. Other system components such as memory and storage may also support multiple modes of operation with trade-offs between power and performance. For example, manufacturers have developed DRAM chips that transition to one of a number of sleep states when idle[33]. As in the case of CPU idle states discussed above, deeper sleep states have greater transition overheads but also save more power. Also, a number of recent papers explore the application of DVS to memory. DVS in DRAM chips is discussed in detail in [30].

2.2 Real-time Scheduling

For all the commercial solutions mentioned in the previous section, there is some form of software that actively monitors the CPU utilization and workload in the system and applies some heuristics to determine the best mode of operation for the system. While this approach is appropriate for non-real-time systems, a different approach is needed for real-time systems.

Real-time systems consist of one or more tasks that share limited CPU time and have timing constraints. To ensure that task timing constraints are met, the available CPU time must be sufficient to accommodate all competing tasks and the tasks must be appropriately scheduled. The minimum CPU-time required for tasks to meet timing constraints depends not only on the CPU usage of the tasks, but also on the scheduling algorithm. When a power-management mechanism trades off performance for lower power, tasks require greater CPU time to perform the same jobs. The trade-off in performance must not be so excessive as to compromise the schedule

and prevent tasks from meeting timing constraints. In the remainder of this section, the periodic real-time task model is presented and relevant scheduling algorithms are described. Power management of real-time systems is described in greater detail in the following section.

The periodic real-time task model was first presented in the Liu and Layland analysis, [50]. A real-time task τ_i, consists of a sequence of bursts of computation called jobs or activations, $J_{i,1}$, $J_{i,2}$, ..., $J_{i,j}$. Each job has a release time $r_{i,j}$ such that the job can be performed *only after this release time* and can be performed *completely without blocking* after the release time. Also, each job must complete before a deadline, $d_{i,j}$. The exact execution time for a job is denoted $c_i[j]$ and the worst-case-execution time (WCET) for the task is denoted C_i. It is generally assumed that the WCET is known.

For periodic real-time tasks, the release times of successive jobs are separated by a constant task period, T_i, such that $r_{i,j+1} = r_{i,j} + T_i$. It is generally assumed that the job deadline relative to the release time is equal to the task period, $d_{i,j} = r_{i,j} + T_i$. Equivalently, the job deadline is assumed to be equal to the release time of the following job, $r_{i,j+1} = d_{i,j}$. For the purposes of classical scheduling algorithms, a task is completely defined by the corresponding task period and WCET. Aspects of the periodic-real-time task model are presented in Figure 2 for clarity.

The Liu and Layland analysis also presents two key algorithms: Earliest Deadline First (EDF) and Rate Monotonic (RM). These algorithms address the problem of scheduling two or more periodic real-time tasks on a single processor and have been used as the basis for some of the reservation-based schedulers and power-management algorithms discussed later in this chapter. Rate-Monotonic is a static-priority scheduling algorithm where tasks are assigned priorities in order of the task periods. The task with the shortest task period is assigned the highest priority. Earliest Deadline First is a dynamic priority scheduling algorithm where tasks are assigned priorities

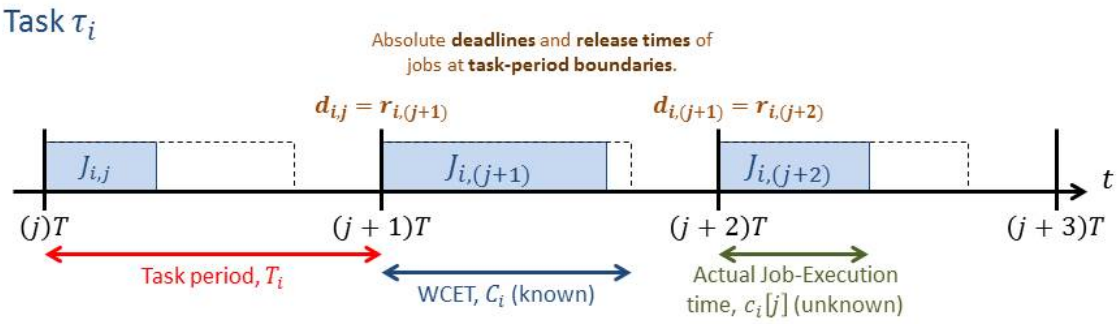


Figure 2: The periodic-real-time task model

dynamically based on time remaining until the next deadline. The task with the earliest deadline is assigned the highest priority.

If a given scheduling algorithm is able to schedule a set of tasks such that all job deadlines are met, the task set is said to be *feasible* under that scheduling algorithm. A *schedulability test* is used to determine if a task set is feasible. In the case of EDF and RM, the schedulability test is based on a value called the utilization factor. The utilization factor, U, for a set of N tasks is defined as follows:

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i}. \tag{2}$$

It is shown in [50] that a task set is feasible under EDF or RM if the utilization factor is less than or equal to some *least upper bound* (LUB). The LUB is 1.0 for EDF and a function of the number of tasks for RM (around 0.7 in the limit). It is further shown that RM is an optimal fixed priority algorithm and that EDF is an optimal dynamic priority algorithm. Optimality implies that any task set that is feasible under some algorithm in the same class(static or dynamic priority) is also feasible under RM and EDF.

EDF and RM are intended for process control applications. While the assumptions about the tasks stated above are appropriate for control applications, they are restrictive for other applications. Papers that followed the Liu and Layland analysis

present ways to relax some of the above constraints. For example, EDF and RM can be modified to handle aperiodic tasks by replacing the task periods in the computation of the utilization factor by minimum inter-arrival times. Some concept of minimum inter-arrival time and worst-case execution time is necessary to have an upper bound on the rate of demand for CPU time.

2.3 Dynamic Voltage and Frequency Scaling

A task set is schedulable if the utilization factor is no greater than the least upper bound corresponding to the scheduling algorithm. When the utilization factor is less than the LUB, the difference in the LUB and utilization factor represents excess CPU capacity called *slack*. Most power-management algorithms for real-time systems are based on exploiting this slack: trading off just enough performance for low power consumption such that the slack is reduced and the schedulability condition is still satisfied.

Also, previous work on power management of real-time systems have been based primarily on DVS and based on certain assumptions about the managed system. It is assumed that CPU power dominates total system power and that dynamic switching power dominates total CPU power. It is further assumed that the CPU-clock speed can be set to arbitrary positive values up to some maximum and that execution times scale inversely with CPU-clock speed. Specifically, if the execution time of a job is c at the fastest CPU-clock speed, then the execution time is (c/α) when the CPU-clock speed is scaled by some factor $\alpha \in [0, 1)$. Lastly, it is assumed that the overhead of transitioning between CPU frequencies is negligible.

The simplest power-management algorithm entails exploiting static slack and is referred to as Static RT-DVS in [60]. Let $\alpha \in [0, 1)$ denote the scaling factor for the CPU clock speed. Let C_i denote the WCET of a task τ_i when $\alpha = 1.0$ and the CPU is configured to run at maximum speed. Given the above assumptions, the scaled

WCET of τ_i is then (C_i/α). Substituting the WCET values in (2) with the scaled WCET values yields an expression for the scaled utilization factor, U_{scaled}:

$$U_{scaled} = \sum_{i=1}^{N} \frac{(C_i/\alpha)}{T_i}. \tag{3}$$

A task set is schedulable under EDF or RM if the utilization factor is less than the LUB corresponding to EDF or RM, respectively. A smaller value of α corresponds to a slower clock speed. As α decreases, U_{scaled} increases and approaches the LUB. The minimum value of α, denoted α_{min}, that satisfies the schedulability condition can be determined by setting the RHS of (3) equal to the LUB and solving for α. The CPU speed is set accordingly in Static RT-DVS. An expression for α_{min} is shown below:

$$\alpha_{min} = \frac{1}{LUB} \sum_{i=1}^{N} \frac{C_i}{T_i} = \frac{U}{LUB} \tag{4}$$

2.3.1 Dynamic Slack Reclamation

Static RT-DVS exploits *static slack* which is computed based on WCETs. However, as shown in section 2.4, job-execution times can vary. The actual execution time of a job may be significantly lower than the WCET for the corresponding task. Jobs that complete using less CPU time than the WCET expose additional unused CPU capacity called *dynamic slack*. Dynamic slack reclamation entails exploiting dynamic slack by reducing the CPU speed further as jobs complete early.

Cycle-conserving RT-DVS (CC RT-DVS) is a slack reclaiming DVS algorithm presented in [60]. There are two variations of the algorithm for the EDF and RM schedulers. In the case of the EDF scheduler, the utilization-factor is recomputed on each job completion. The new utilization factor is computed by replacing the WCET for the corresponding task with the actual execution time of the newly completed job. Given the reduced utilization factor, the CPU speed can be reduced further according to (4). When a new job is released, the utilization factor must be computed again

with the WCET since the actual execution time is no longer known. The CPU speed is increased according to the new utilization factor. In the case of the RM scheduler, the speed is still adjusted on job completion and job release but the adjustment is performed differently as described in detail in [60].

Given the higher LUB associated with the EDF algorithm, power-management algorithms for real-time systems generally perform better with the EDF scheduler. Another slack reclaiming DVS algorithm based on EDF is the Generic Dynamic Reclaiming Algorithm (GDRA) presented in [14]. The general approach in GDRA is to initially set the CPU clock speed in the same way as in the Static RT-DVS algorithm. A canonical schedule is defined as the schedule that would result from all jobs running with corresponding WCETs and the CPU running at the Static RT-DVS speed. As jobs complete earlier than anticipated in the canonical schedule, the exposed slack is used to slow the CPU further and save additional power. The canonical schedule is maintained using a data structure called α-queue. Simulation results show that GDRA performs better than the CC RT-DVS algorithm as discussed in detail in [14].

2.3.2 Accelerating Voltage Schedules

Both CC RT-DVS and GDRA initially set the CPU to a high clock speed to accommodate the WCET and then reduce the clock speed as jobs complete early. An alternative to this decelerating approach is an accelerating approach that entails deferring work as much as possible. In such an approach the CPU speed is initially set as low as permitted by the schedulability condition and then accelerated to allow jobs to complete before the deadline. However, if jobs require significantly less CPU time than the WCET, it may not be necessary to run the CPU at the higher speeds. Not having to run at the higher speeds could yield additional power savings.

Look-Ahead RT-DVS (LA RT-DVS) is an EDF-based accelerating DVS algorithm presented in [60]. Tasks are allocated CPU time in reverse-EDF order starting with

the task with the latest deadline. The execution of each task is deferred as far as possible without causing the task to miss the corresponding deadline. This operation exposes additional slack before the deadline of the earliest-deadline task. This slack is used to reduce the CPU speed and reduce power consumption. The CPU-speed is recomputed in this way on each job completion and each job release. The details of this algorithm are presented in [60].

Another EDF-based accelerating DVS algorithm is Aggressive-DR[14]. Aggressive-DR is designed to reduce power consumption in the common case. Initially, the CPU speed is set to the Static DVS speed computed based on WCETs. An additional static optimal speed, S_{optavg}, is computed assuming that the actual execution time of all jobs are equal to the average-case execution time and not the WCET. An attempt is made to transfer CPU time from later-deadline jobs to the earliest-deadline job such that the earliest-deadline job can be run at the slower S_{optavg} speed. However, since the CPU time allocated to later jobs is now reduced, those jobs may have to run at higher CPU speeds to complete in the shorter time. The CPU time transferred from later-deadline jobs is never so large as to compromise the schedulability of those jobs at the maximum CPU speed. On the other hand, if the actual execution time of the earliest-deadline job is less than or equal to the average-case execution time, the CPU can continue to run at lower speeds without compromising schedulability. According to simulation-based results, Aggressive-DR performs better than LA RT-DVS. The details of this algorithm are presented in [14].

2.3.3 Intra-Task DVS

CC RT-DVS, LA RT-DVS, GDRA, and Aggressive-DR are all *inter-task* DVS algorithms that adapt the CPU-clock speed at job boundaries: release time, completion time, and deadline. On the other hand, *intra-task* DVS algorithms adapt the CPU-clock speed during job execution. All intra-task DVS algorithms are accelerating DVS

algorithms.

An example of intra-task DVS is *Feedback-DVS* [75]. Feedback-DVS splits each job into two halves. The first-half execution time is the predicted job-execution time while the second-half execution time is the difference between the WCET and first-half execution time. Job-execution times are predicted using a PID controller. The second half is scheduled to execute at the maximum possible clock speed whereas the first half is executed at the minimum clock speed that allows timely job completion. Provided that the predicted execution time is greater than or equal to the actual execution time, the second half is never executed and the clock speed is kept low. Simulation results presented in [75] show that Feedback-DVS generally performs better than LA.

Some papers have proposed more fine-grained intra-task DVS to minimize expected CPU energy [37, 23, 70, 24, 61]. The general approach is to assume that every job consists of a number of cycles, $c[j]$, up to some maximum, C. While the exact number of cycles required per job is unknown, it is assumed that the probability distribution of the number of cycles is known. Also, it is assumed that the number of cycles required to execute a job is independent of the CPU-clock speed. In theory, the intra-task DVS algorithms can determine the optimal speed to execute each job cycle such that the total expected CPU energy is minimized and all job deadlines are met. However, since adapting the CPU speed for every cycle is not feasible, the distribution of required execution cycles is divided into coarser bins, and an optimal speed is determined for each bin.

Since intra-task DVS algorithms are accelerating algorithms, they require static slack. Different intra-task DVS papers differ on how static slack is distributed between tasks, how dynamic slack is reclaimed, and how the voltage schedule is optimized. Also, more recent papers address intra-task scheduling on real-world platforms with discrete CPU speeds [23, 70, 61] and non-negligible static power [24]. Power management with discrete CPU speeds and non-negligible static power is addressed in

greater detail in Section 2.5.

2.4 Exploiting Soft Timing Constraints

The scheduling and power-management algorithms discussed in Sections 2.2 and 2.3 are aimed at hard-real-time applications. Hard-real-time applications are not allowed to miss job deadlines, and any missed deadline constitutes an error or failure in the system. To ensure that jobs always complete before corresponding deadlines, EDF, RM and related power-management algorithms ensure that sufficient CPU capacity is available to accommodate the WCETs of jobs from all tasks. However, execution times can have a lot of variability, and WCETs are often significantly larger than the average-case execution times.

To illustrate this variability, execution times were measured for decoding successive frames of 10 different high-definition (1080p) MPEG4 video files. The maximum, mean, and minimum execution times are plotted in Figure 3. The chart shows that for all the videos considered, job-execution times have a large range. In the worst case (v_{10}), the frame decoding times range from 4.3ms to 15.0ms. The average utilization for a WCET-based allocation scheme can be measured as the ratio between the mean execution time and the WCET. In the execution times presented in Figure 3, the highest average utilization is 63% (v_1). The lowest average utilization is 42% (v_7). In all the cases, more than 35% on average of the CPU time allocated to these tasks would go unused. Similar reports of variability in execution time and severe and persistent under-utilization of allocated CPU time for another MPEG decoding workload is presented in [5].

In DVS-based power-management algorithms, variability in execution time and resulting dynamic slack is exploited using techniques such as slack reclamation and accelerating CPU speeds. However, in most CPU power models, there is a convex relationship between clock frequency and power dissipation: executing at double the

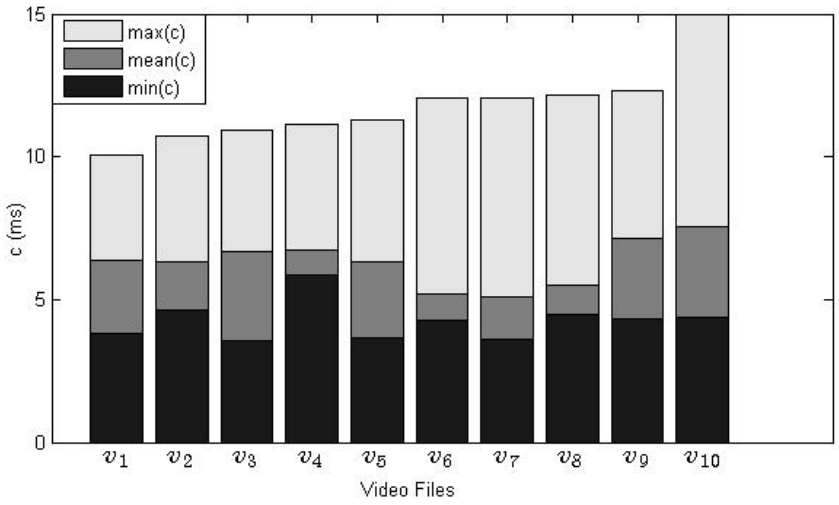


Figure 3: Maximum, mean, and minimum decoding times of 10 high-definition MPEG4 video files.

CPU frequency can more than double the power consumption. Therefore, running jobs at a constant moderate CPU frequency can result in lower power dissipation than running the same jobs in a combination of high and low frequencies, even if the number of cycles executed in these two cases are the same.

In both slack reclaiming and accelerating DVS algorithms, the need to run the CPU at higher clock frequencies is to accommodate rare jobs with worst-case execution times. It may be possible to achieve greater power savings by relaxing the hard timing constraint and allowing such jobs to miss corresponding deadlines. Many applications can tolerate occasional violations of timing constraints and missed deadlines at the cost of degradation in the quality of service. Such applications are described as being soft real time (SRT). This soft-real-time task model can be applied to supervisory control and data acquisition (SCADA) as described in [57]. Also, this task model is applicable to multimedia playback and recording applications that are common in personal mobile devices [5]. In such applications, accommodating the WCET can lead to better timeliness than required and lower CPU utilization and greater power dissipation than desired.

2.4.1 CPU Reservations

To be able to allocate less CPU time to jobs than the corresponding WCETs, appropriate policies and mechanisms are needed to handle the case where a job execution time is greater than the allocated CPU time. EDF and RM scheduling algorithms require the use of WCET because a task that executes for longer than anticipated can compromise the schedule of all lower priority tasks. With reservation-based schedulers, each task or group of tasks is explicitly allocated some amount of time to run on the processor. The allocation is enforced through the use of timers and preemption. Such schedulers offer *temporal isolation* where a task that requires more execution time than anticipated is not able to consume or interfere with the time allocated to other tasks.

The concept of CPU capacity reserves was first presented in [53] in the context of scheduling multimedia applications in a general purpose OS. A reserve for a task τ_i consists of a budget, Q_i, and a reservation period, T, such that τ_i is guaranteed to receive Q_i units of CPU time every T units of time. As a task runs, the budget decreases. When a task depletes the allocated CPU time, the budget reaches zero and the task is throttled, put to sleep until the following reservation period. At the beginning of every reservation period, the budget is replenished and the task is unthrottled. This throttling behavior allows jobs to have longer execution times than anticipated without compromising the schedule of other tasks.

For a non-periodic task, the budget and reservation period are assigned based on delay requirements. This is described in greater detail in [53]. For a periodic real-time task, the reservation period, T, is set to the task period, T_i, or some sub-multiple of the task period, such that $T_i = KT \; \exists K \in \mathbb{Z}^+$. Tasks are scheduled according to the EDF or RM algorithm, but execution times are replaced by allocated budgets and task periods and relative deadlines are set to the reservation period. The schedulability tests discussed in Section 2.2 can be used to determine if a set of reservations are

feasible. Mercer et al. discuss using such schedulability tests for admission control [53].

Given that the budget allocated to a task can be less than the corresponding WCET, an important consideration is how to handle overrun conditions. A job is said to be *overrunning* if the job-execution time is larger than the budget allocated to the corresponding task ($c_i[j] > Q_i$). Abeni and Buttazo propose a reservation model where an overrunning job is allowed to continue execution using the budget from later reservation periods and complete after the deadline [4]. Depending on the length of the overrunning job, one or more successive jobs may be queued. This is referred at as the *queuing SRT-task model*. Abeni and Buttazo show that if the allocated budget is greater than the mean execution time, job-queue lengths are stable.

The approach proposed by Abeni and Buttazo is intended to schedule HRT tasks and SRT tasks on the same processor. HRT tasks are scheduled according to the EDF algorithm. Sufficient CPU time is reserved to accommodate the WCET of jobs from HRT tasks. SRT tasks are associated with Constant Bandwidth Servers (CBS) and the CBS are scheduled with HRT tasks by the same EDF scheduler.

A CBS is similar to a reserve as described above. The relative deadline, $d_i[k]$, for a CBS is equal to the corresponding reservation period, $d_i[k] = r_i[k] + T$. When a task associated with a CBS runs, the CBS budget decreases. However, when the CBS budget reaches 0, the budget is immediately replenished and the CBS deadline is delayed by the reservation period. Because a CBS is never throttled as described earlier, this approach is *work conserving*; the CPU is never idle while jobs are incomplete. The CBS approach naturally reclaims unused budget from tasks with jobs that finish early and assigns that budget to tasks with overrunning jobs.

A number of other reservation-based schedulers have been presented in previous work. GRUB (Greedy Reclamation of Unused Bandwidth) is an algorithm built on CBS [49] that more efficiently redistributes unused budget between running tasks.

Lindberg presents a survey of other reservation-based schedulers in [48].

2.4.2 Adaptive CPU-Budget Allocation

In both the CBS and GRUB algorithms, the budget allocated to a task is static. As mentioned above, this budget must be strictly greater than the mean job-execution time to ensure a stable job queue. If the probability distribution of job-execution times is static and known, budget can be allocated such that a job will meet the corresponding deadline with some probability [4]. However, job-execution times and distributions are not known a priori for most applications, and execution times vary as shown at the beginning of this section. There is a need to adapt the budget allocated to SRT tasks based on dynamic workload requirements.

Abeni et al. present the *Adaptive Reservations* abstraction in [6] where the allocated budget is adapted on completion of every job based on the *scheduling error* of the completed job. Scheduling error is defined as the difference between the deadline and the latest possible finishing time (LFT) or the *virtual finishing time* (VFT) which are described in greater detail in Section 3.1.2 of Chapter 3. A positive scheduling error is an indication of insufficient budget allocation and poor timeliness, whereas a negative scheduling error is an indication of excess budget allocation and poor budget utilization. Abeni et al. present a model to describe the dynamics of scheduling error under Adaptive Reservations and propose a PI controller to drive the scheduling error to zero.

Job-execution times have some degree of randomness that is not directly addressed in [6]. Song et al. present a slightly different model for the same VFT-based system and present a controller design to handle time varying uncertainty in job-execution times [67]. Cucinotta et al. also address the problem of adaptive reservations, but present a stochastic model[27]. The sequence of execution times from a task is modeled as an independently and identically distributed (i.i.d.) random process. Budget

allocation decisions are made based on the VFT of previous jobs as well as predictions of future execution times.

Another approach to adaptive budget allocation is to maintain a moving window of the execution times of recently completed jobs. These execution times may be sorted or binned to obtain a desired percentile, P, of past execution times. In the case where no previous jobs are queued and the binned or sorted execution times accurately approximate the probability distribution, budget allocation based on the P^{th} percentile should result in a deadline miss-rate equal to P. Inherent in this approach is the assumption that the distribution does not change with time and that execution times of successive jobs are i.i.d. However, an advantage of this approach is that there are no parameters that have to be tuned in the way that gains have to be tuned for control systems. This type of approach is adopted in GRACE-OS [72] and the DVS system presented in [26]. These systems are described in greater detail in Section 2.4.3.

2.4.3 Power-Management of Soft-Real-Time systems

The power-management algorithms described in Section 2.3 that are applied to HRT applications may be applied to SRT applications as well but with key differences. To take advantage of the soft timing constraints, reservation-based schedulers are used and the CPU budget allocated to the SRT tasks are less than the WCETs. Furthermore, the WCET values and task periods in Equation 4 are replaced with the budget allocations and reservation period, respectively. Since the budget allocation is smaller than the WCET, the resulting value of α_{min} is smaller, and the CPU speed can be reduced further. Also, with a budget allocation smaller then the WCET, the average dynamic slack is reduced as well, and the benefits of dynamic slack reclamation or an accelerating voltage schedule may be diminished. In the remainder of this section, three different systems are considered for the power-management of

platforms hosting soft-real-time applications.

The first is the GRACE-OS system presented in [72]. The system consists of three major components: a profiler, an SRT scheduler, and a speed adapter. The *profiler* maintains a history of the number of cycles required to execute recent jobs. This history is used to build a histogram to approximate the probability distribution of job-execution cycles. The *speed adapter* computes a frequency schedule using the histogram built by the profiler. The approach used to compute the frequency schedule is similar to the approach used in the fine-grained intra-task policy described in Section 2.3.3. However, the speed schedule only accommodates up to the ρ^{th} percentile cycles rather than the worst-case execution cycles, where ρ is a QOS parameter denoting the probability that a job will meet the corresponding deadline. To allow for cycle allocations that are less than the worst case execution cycles, GRACE-OS uses a reservation-based scheduler built on EDF, with budget allocations based on cycles rather than time.

GRACE-OS has been further extended in later work. A more practical design of GRACE-OS is discussed in [73] to address non-ideal properties of real platforms. Real-world platforms are discussed in greater detail in Section 2.5. Also, GRACE-OS has been extended to allow joint application adaptation and CPU-speed adaptation in the systems presented in [71] and [69].

In GRACE-OS, only timing guarantees and power savings are addressed. An alternative system is presented in [26] that also takes into consideration the QOS delivered by the hosted applications. A reservation-based scheduler is used, and the allocated budget is adapted to guarantee a probabilistic upper bound on the scheduling error. In addition to the modes of operation supported by the platform, the managed applications may be configured to different modes as well. The platform mode affects power usage whereas the application mode affects the delivered QOS. Both the application mode and platform mode affect the budget usage. The budget

allocation is adapted at a faster rate in an inner loop, whereas the modes of applications and platform are adapted at a slower rate in an outer loop. The configuration of the modes of the application and platform is posed as a binary-linear-programming problem with a goal function based on the delivered QOS and power usage. This optimization operation requires access to a matrix of minimum budget allocation for each combination of the application mode and platform mode, and this matrix is required for each application. Details of this approach are presented in [26].

DVFS-RBED is another DVFS system for soft-real-time applications [47]. In this system, frequency scaling is applied not only to the CPU, but also to memory, bus, and IO clocks. This system is built on a reservation-based scheduler called RBED that is very similar to the CBS algorithm. Unlike GRACE-OS or the reservation-based scheduler proposed in [26], the budget allocated to the different tasks are not adapted at runtime. At each scheduling event, the frequency set point for the preempting task is computed using a linear search through the discrete set of available frequencies. The chosen frequency is the one with the lowest power consumption that still allows jobs from the preempting task to complete on time. Whenever a job consumes less CPU time than the allocated budget, the dynamic slack is donated to the next runnable task. This slack is used to slow the CPU further and save additional power. The power consumption and performance degradation associated with each frequency is predicted using adaptive models. These models are discussed in greater detail in Section 2.5.5.

2.5 Real-World Platforms

Most DVS algorithms are designed with assumptions that are not valid on real-world platforms. Except in CPU-bound applications, job-execution times are not inversely proportional to CPU-clock frequency. Dynamic CPU power does not always dominate total CPU power. The hardware might not allow the CPU-clock frequency to be set

to arbitrary values; the power-management algorithm may have to choose from a discrete set of DVS settings. Parameters related to the power and performance of the CPU at different clock frequencies are not always static. Lastly, the overhead of transitioning to sleep states is not always negligible. The following subsections relate to previous work on how to addresses some of the non-ideal behaviors of real-world platforms.

2.5.1 On-chip Work and Off-chip Work

Firstly, job-execution times do not always scale inversely with CPU-clock frequency [12, 65]. Execution times have on-chip and off-chip components. The off-chip component of execution time consists of I/O operations and memory operations that go off chip due to cache misses. While the on-chip component of execution time scales with clock frequency, the off-chip component does not. This modified relationship between CPU frequency and execution time is expressed in (5).

$$c(\alpha) = \frac{c_{\text{on-chip}}}{\alpha} + c_{\text{off-chip}}. \tag{5}$$

Execution time as a function of the scaling factor is denoted $c(\alpha)$. The scaling factor, α, is a ratio of the target clock frequency and maximum clock frequency such that $\alpha = (f/f_{max})$. The variable, $c_{\text{on-chip}}$ denotes the on-chip component of execution time at the maximum clock frequency. The variable, $c_{\text{off-chip}}$ denotes the off-chip component of execution time. Based on (5), the number of CPU cycles required to complete a job is not constant across CPU frequencies. Furthermore, given a fixed amount of slack, the modified performance model allows CPU clock frequency to be reduced further than what is thought possible with the simplified linear model.

Accurate prediction of performance degradation requires knowledge of $c_{\text{on-chip}}$ and $c_{\text{off-chip}}$. Some applications are CPU bound and others are memory bound. Therefore, $c_{\text{on-chip}}$ and $c_{\text{off-chip}}$ are application specific. Amur et al. show how $c_{\text{off-chip}}$ is related to events in the memory hierarchy such as miss rates in the last-level on-chip cache

[12]. Such events can be measured with hardware performance-monitoring counters (PMCs). Dynamic estimation of $c_{\text{off-chip}}$ is discussed in greater detail in Section 2.5.5.

2.5.2 Leakage Power and Critical Frequency

The second invalid assumption is that dynamic switching power dominates total CPU power. Static power is becoming increasingly significant due to shrinking process technology and decreasing supply voltage. As a result, a system that just minimizes dynamic CPU power can consume more total power than a system that runs constantly at maximum speed. A system running at maximum speed can complete the required computation faster and enter a sleep state to reduce static power. On the other hand, a traditional DVS algorithm eliminates all idle time and eliminates the possibility of entering a sleep state.

Jejurikar et al. present the concept of *critical frequency* in [43]. Critical frequency is the clock frequency where the total CPU energy dissipated per cycle is minimized. Jejurikar et al. suggest that it is more energy efficient to run the CPU at the critical frequency and transition to a sleep state than to run the CPU at a speed below the critical frequency. Below the critical frequency, static power dominates total power. The existence of a critical frequency is confirmed by the experimental results from a real embedded platform[64]. Snowdon et al. show that for a set of fixed workload, measured CPU energy is lower than what is predicted by a simple model based on dynamic CPU energy alone. For sufficiently low frequencies, reducing the CPU frequency further results in an increase in total CPU energy.

Also, if a task set is feasible when executed below the critical frequency, it is feasible at the critical frequency. Therefore, any DVS algorithm can be trivially modified such that only speeds at or above the critical frequency are used. The concept of critical frequency is applied to a slack-reclaiming DVS algorithm in [15]. The critical frequency is computed per task using the modified performance model

described by (5) and an energy model that includes static CPU power. This task-specific critical frequency is shown to be a function of the ratio of on-chip to off-chip execution time and other parameters. A static CPU frequency greater than the critical frequency is assigned to each task such that total energy is minimized and all timing constraints are satisfied. Finally, a slack reclaiming algorithm is applied during run time. This slack reclaiming algorithm is built on GDRA mentioned in Section 2.3.1, but takes the critical frequency and off-chip work into account when reclaiming slack.

The concept of critical frequency is a simple approach towards the power-management of platforms with non-negligible static power. However, the critical-frequency approach does not take into account the overhead of transitioning to a sleep state.

2.5.3 Sleep States and Break-even Time

In classic DVS work, it is assumed that the overhead of transitions between operation modes are negligible. The validity of this assumption depends on the nature of the state transition and how frequently the transitions take place. Transition time between processor DVS settings may be negligible. On the other hand, transitions to and from deep sleep states can take much longer. For example, on Intel processors, transitions to the C3 or C6 sleep state result in loss of cache content. Therefore, as described in Section 2.1, transition to a sleep state is only possible and worthwhile if the CPU-idle time is longer than the minimum transition time and break-even time.

Furthermore, other components such as DRAM may also have sleep states. The minimum transition time and break-even time for these additional components may be different from those of the processor. Given the different break-even times of different components, the critical frequency cannot be computed in the same way as when considering only CPU static power. Devadas and Aydin address this problem of minimizing system-wide energy using both DPM and DVS, taking into account the break-even times of different components [31]. However, managing sleep-states of

non-CPU components is beyond the scope of this dissertation.

2.5.4 Discrete DVS Settings

Another invalid assumption is that the CPU-clock frequency can be set to arbitrary positive values up to some maximum. In practical frequency-scaling mechanisms, such as Intel SpeedStep and AMD PowerNow!, the CPU-clock frequency must be chosen from a discrete set of values. The simplest way to address power management with discrete DVS settings is to select the lowest frequency greater than the desired frequency. However, this can lead to high energy consumption [21].

A more common approach is to periodically alternate between the two available frequencies that are closest to the desired frequency. The time spent in each of these two frequencies must be chosen such that the time-weighted average of the two frequencies is equal to the desired frequency. With simple convex power models that disregard transition overheads, this approach is optimal.

While the overhead of changing the CPU-clock frequency may be negligible, the relationship between CPU energy and CPU-clock frequency may not be convex. To address this problem, Dabiri et al. present a processor model with generic discrete modes of operation[29]. These operation modes offer a trade-off between power and speed, but no assumption is made about the relationship between power and speed. Given this model, the processor can still be run at arbitrary speeds on average by periodically alternating between different operation modes. However, the pair of operation modes that minimizes CPU energy for the given speed is not as easily determined.

In the algorithm presented in [29], the first step is to extract a subset of operation modes from the set of all operation modes for which the convex combinations of power-speed points make up a *lower convex curve*. The time complexity of extracting this subset of operation modes is $O(n \log n)$, where n is the number of operation

modes. It is shown that the most energy efficient way to run the processor at a desired speed on average is to run the processor at no more than two consecutive operation modes from the extracted subset of operation modes. The complexity of the complete algorithm, which also addresses transition overheads, is shown to be $O(n^2 \log n)$.

2.5.5 Dynamic Power and Performance Parameters

Power management of applications that are not CPU bound requires knowledge of the component of execution time that is due to off-chip work. Addressing leakage power requires knowledge of the critical frequency. Both off-chip work and critical frequency are application-specific parameters. Snowdon et al. argue that applications and corresponding workloads are dynamic, and that power and performance characteristics cannot be profiled off line [66]. Instead, these parameters must be learned dynamically.

Snowdon et. al. present a performance model where execution time is a function of bus frequency, memory frequency, and CPU frequency [65] as shown in (6). The proposed system is primarily intended for a specific embedded platform where these frequencies are configurable.

$$c = \frac{C_{CPU}}{f_{CPU}} + \frac{C_{bus}}{f_{bus}} + \frac{C_{mem}}{f_{mem}} + \frac{C_{io}}{f_{io}} + ... \tag{6}$$

Total job-execution time is denoted c. CPU clock cycles and bus clock cycles are denoted C_{CPU} and C_{bus}, respectively. The number of clock cycles for other components are similarly denoted. As in the case of (5), the number of cycles C_{CPU} and C_{bus} are application specific.

Snowdon et al. show that the parameters C_{CPU} and C_{bus} are correlated to architectural events that can be counted using PMCs and that this correlation is platform specific. Furthermore, it is assumed that the rate of occurrence of these architectural events varies slowly or do not vary for a given application. This assumption is based

on the principle of temporal locality. A system is proposed where the coefficients relating C_{CPU} and C_{bus} to the PMC-counted events are determined off-line. These coefficients are then used on-line with PMC-based measurements to estimate C_{CPU} and C_{bus} and predict the performance degradation of applications due to frequency scaling. Experimental results are presented that show that the prediction algorithm works well.

A similar approach is applied to the prediction of power consumption in [66]. An energy model is presented that relates consumed power to architectural events and CPU, memory, and bus frequencies. As in the case of the system for performance prediction, architectural events are counted using PMCs, and the model coefficients that relate power consumption to component frequencies and architectural events are computed off line. The model is then used on line with PMC readings to predict power consumption at different component frequencies. These two approaches for predicting power consumption and performance degradation are applied to a real-time system called DVFS-RBED as discussed in Section 2.4.3.

In [65] and [66], the coefficients used to predict performance degradation and power consumption from architecture events are assumed to be static and determined off-line. It is possible to learn these relations dynamically during runtime using recursive-least-squares (RLS) filters and Kalman filters as discussed in [74]. However, the disadvantage of using RLS filters and Kalman filters is the higher overhead.

2.6 Comparison of LAMbS to Previous Work

The Linear Adaptive Models-based System has a number of advantages over the power-management systems presented in previous work.

In LAMbS, the platform is modeled as having a discrete set of modes of operation with arbitrary performance and power consumption rate in the same way as in [26] and [29]. This platform model eliminates a number of the invalid assumptions.

Assumptions about linear degradation in performance with CPU frequency are no longer necessary. Assumptions about convex relationships between CPU frequency and power dissipation are no longer necessary. Assumptions about being able to set the CPU clock to arbitrary frequencies are no longer necessary. Furthermore, modeling the hardware as having discrete modes of operation allows for a more general framework for power management where more exotic power-management mechanisms may be used such as the big.LITTLE model discussed in Section 2.1.1.

Second, LAMbS dynamically monitors and adapts to changes in the rate of power dissipation and performance associated with each mode of operation. This dynamic adaptation eliminates the need for off-line tuning of power and performance parameters such as the off-chip component of execution time. DVFS-RBED, described in [47], dynamically monitors architectural events and uses such events to estimate the different components of execution time in the power and performance models. However, DVFS-RBED still requires calibration of the coefficients that relate the architectural events to estimates of execution-time components. In contrast, the adaptive approach proposed in LAMbS does not require any tuning. Eliminating the need to tune parameters allows the system to be more practical.

Third, LAMbS allows CPU budget allocations to be adapted dynamically to the changing load of hosted applications. While budget adaptations have been widely used in previous work, the PBS algorithm allows budget allocations to be adapted at a faster rate. Furthermore, power-management decisions are closely tied to the CPU budget committed to different tasks. Therefore, faster adaptations of budget allocations allow power-management decisions to be made at a faster rate.

Furthermore, the two-stage predictor built on the PBS algorithm allows for the use of arbitrary first-stage predictors to improve the efficiency of budget allocation. The predictors are per task and can be based on application-specific domain-knowledge or just correlation in the execution-time of successive jobs. An accurate predictor would

reduce uncertainty in the budget allocation required for each job and improve budget utilization without an increase in the deadline-miss rate. Better budget utilization can help to reduce wasted CPU capacity and reduce power consumption.

Finally, a novel measure of computation called VIC is used for budget allocation that decouples and simplifies the problem of reservation-based scheduling and power management. For example, it is not necessary for the CPU budget to be specified separately for every mode of operation as done in [26]. Also, it is not necessary to use a static model to relate the degradation in performance to the choice of mode of operation as typically done in power-management systems based on frequency scaling such as [72].

Overall, LAMbS is a robust, autonomic, and practical system for power-management that is applicable to real-world platforms.

CHAPTER III

ASYNCHRONOUS BUDGET ADAPTATIONS

An essential component of LAMbS is the CPU budget-allocation mechanism. The performance of this mechanism has a significant impact on overall energy savings. If the budget-allocation mechanism persistently under-allocates budget to a task, the amount of CPU capacity reserved is less overall. Lower budget allocations can lead to greater energy savings but at the cost of more jobs missing corresponding deadlines. On the other hand, the budget-allocation mechanism could provision a task with sufficient budget to allow a larger percentage of jobs to meet corresponding deadlines. However, depending on the level of uncertainty in job-execution times, provisioning tasks to meet deadlines more often may require greater levels of budget over-allocation that can lead to greater energy consumption. Therefore, the performance of a budget-allocation algorithm is measured in terms of both the average budget allocation and the resulting timeliness of jobs. It is desirable to have an algorithm that can deliver the same deadline-miss rate with lower average budget allocation.

The obstacle to accurate budget allocation is uncertainty in job-execution times; the execution time of a job is not known until that job completes. The execution time of a job may be affected by various factors such as the number of interrupts that occur during execution or the level of cache pollution resulting from other tasks sharing the processor. It may be difficult to determine an appropriate budget allocation off-line. By adapting the budget allocation dynamically based on real-time measurements of CPU usage, it is possible to have a system that is more autonomic and robust against uncertainty.

There has been significant previous work on adaptive CPU-budget allocation.

Khalilzad et al. propose an approach based on controlling CPU utilization and deadline miss rate [44]. One of the problems with miss-rate and utilization-based control is that the response can be slow [11]. On the other hand, as described in Section 2.4.2, finishing time (FT) and virtual finishing time (VFT) can be measured on the completion of every job making VFT-based and FT-based adaptation more responsive than miss-rate and utilization-based approaches. Along the same lines, the PBS algorithm developed in this chapter improves on the responsiveness of previous approaches by performing the budget adaptation at a faster rate. The PBS algorithm has been presented in [8], with additional details of queue-stability and performance with real workloads presented in [7].

In PBS, budget allocations are adapted at reservation-period boundaries based on the estimated *computational load* and the time remaining until the next earliest deadline. The computational load at the beginning of a given reservation-period is the remaining dedicated processing time needed by the corresponding task to complete all jobs released before the start of that reservation period. Assuming the queuing SRT-task model described in Section 2.4.1, the computational load is estimated based on the number of jobs queued, the execution time of recently completed jobs, and the CPU-time already consumed by the currently running job.

To appreciate the advantage of the PBS approach, consider the case of a task that undergoes a significant increase in average job-execution times. Initially, in the case of both the PBS-based approach and VFT-based approach, the budget allocation is based on the CPU usage of previous jobs and the newer jobs with higher execution times are under-allocated. One or more of these newer heavier jobs may miss the corresponding deadlines. However, in the case of the PBS algorithm, the response to the under-allocation occurs as soon as the first missed deadline when a new job is released and queued. In contrast, a VFT-based approach can not detect or respond

to the under-allocation until the first under-allocated job completes. The faster response of the PBS algorithm is confirmed by simulation results presented in Section 3.4. Furthermore, experimental results presented in Section 3.6 show that the PBS algorithm maintains a fair and graceful degradation in the timeliness of managed tasks under overload conditions and that PBS is able to recover rapidly from an overload condition.

The remainder of this chapter is organized as follows: the task model and reservation-based scheduling model is elaborated in Section 3.1; a detailed description of PBS is presented in Section 3.2; a proof of queue stability is presented in Section 3.3; simulation results are presented in Section 3.4; a Linux-based implementation of PBS is described in Section 3.5; finally, experimental results are presented in Section 3.6.

3.1 SRT Tasks and Reservation-based Scheduling

The PBS algorithm and the derivative TSP algorithm presented in Chapter 4 are directed at periodic real-time tasks with soft timing constraints.

Each task, τ_i, consists of a sequential stream of jobs and each job consists of some amount of computation. The computation associated with the j^{th} job, $J_{i,j}$, can be performed completely without blocking after the release time, $r_{i,j}$. Also, a deadline, $d_{i,j}$, is associated with each job. If the computation associated with a job does not complete by the corresponding deadline, the deadline is said to be missed. The release time of successive jobs of a *periodic* task τ_i differ by some constant T_i, that is, $r_{i,(j+1)} = r_{i,j} + T_i$. The time interval between the release of two successive jobs is referred to as a task period. Furthermore, the release time of each job is the deadline of the preceding job, $r_{i,j} = d_{i,(j-1)}$. Therefore, the deadline of a job is one task period from the release time: $d_{i,j} = r_{i,j} + T_i$. The execution time, $c_i[j]$, is the amount of dedicated processing time needed to complete job $J_{i,j}$. This execution time is unknown until the job completes.

As mentioned in Section 2.4, jobs from SRT tasks may occasionally miss corresponding deadlines. When a job does not complete by the corresponding deadline, the job is still allowed to continue execution until completion which may be after the release of one or more successive jobs. Because jobs are executed in order, jobs released before the completion of preceding jobs are queued and executed after the preceding jobs complete.

One possible class of applications that fits the above task model is multimedia processing like MPEG4 video decoding. In MPEG4 decoding, video frames have to be decoded and displayed periodically and decoding a frame requires the previous frame to be decoded. Depending on the content of the video, the execution time needed to decode a frame can vary from frame to frame. Furthermore, this execution time can not be known precisely until the decoding operation completes.

The non-blocking and soft real-time requirements can be met by having a pipelined software architecture with separate software threads implementing stages of the pipeline. I/O operations like reading from disk and writing to a frame buffer can be handled by hard-real-time threads with fixed budget allocations. The brunt of the computation can be performed by a soft real-time decoding thread with adaptive budget allocation. I/O threads are by definition I/O bound and should have minimal and deterministic computational requirements which makes hard-real time scheduling of such threads feasible and acceptable. Also, frame deadlines may be several task periods after the decoding deadline, and the input and output from the decoding thread may be queued in input and output buffers. These buffers should allow the decoding thread some laxity in job completion times and allow the decoding thread to be scheduled as an SRT task.

3.1.1 Reservation-based Scheduling

Both the PBS algorithm and the TSP algorithm apply to reservation-based schedulers. As described in Section 2.4.1, reservation-based schedulers periodically commit some amount of time to each real-time task. Budget allocations are enforced through the use of timers and preemption. Such schedulers offer temporal isolation; a task that requires more execution time than anticipated is not able to consume or interfere with the time allocated to other tasks.

Time is divided into periodic intervals called reservation periods that are integer sub-multiples of the task period. The length of a reservation period is denoted T_R. A budget allocation, $Q_i[k]$, for a task, τ_i, consists of some amount of time reserved over the k^{th} reservation period. As the task runs on the processor, the budget is depleted. When the budget reaches zero, the task is preempted until the $(k+1)^{th}$ reservation period. Budget allocations, as addressed in this dissertation, are *hard* [5] and so the scheduler is non-work-conserving; excess CPU budget from one task is not transferred to an under-allocated task as in [4] and [49]. However, the results presented in rest of this chapter show that the PBS algorithm performs well in the more constrained case of hard allocations. Therefore, it is expected that PBS will perform better without the hard-allocation constraint.

An additional variable of interest is the total computational load mentioned at the beginning of this chapter. The computational load, $L_i[k]$, for a task τ_i in the reservation period k denotes the remaining dedicated processing time needed by τ_i to complete all jobs released before the start of reservation period k. The computational load spikes at every task-period boundary when a new job arrives, and it decreases monotonically between task-period boundaries as the task runs. When all released jobs are completed, L_i reaches zero.

Figure 4 shows an example schedule of two tasks with a constant budget allocation. For simplicity all values are normalized by the reservation-period length, T_R. Tasks

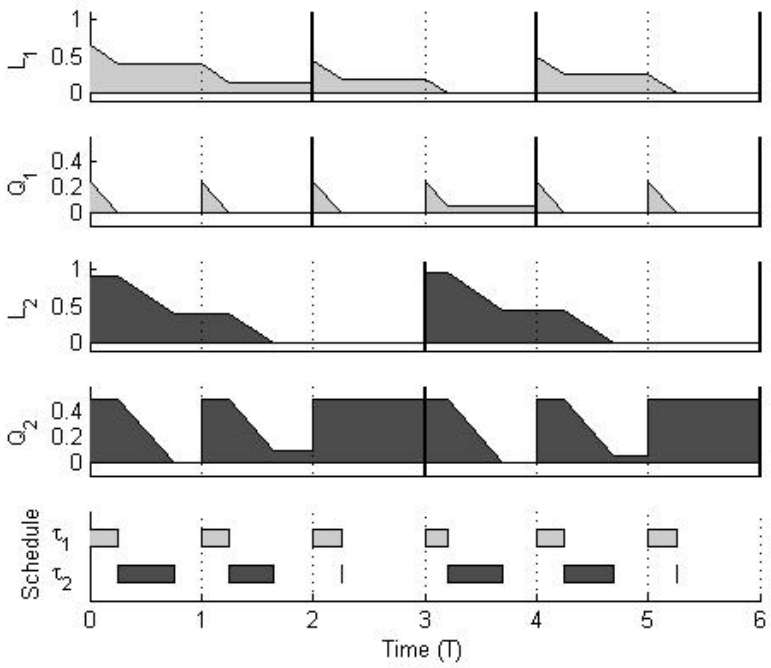


Figure 4: Example schedule of two SRT tasks, τ_1 and τ_2, with a constant budget allocation.

τ_1 and τ_2 have periods of 2 and 3, respectively, and are assigned constant budgets of 0.25 and 0.5, respectively, per reservation period. The reservation-period boundaries are marked by dotted lines, whereas the task-period boundaries are marked by thick solid lines. Only one task can run at a time, and for a given task, the computational load, L_i, and the budget remaining, Q_i, only decrease when that task runs.

Consider the schedule of τ_1 in Figure 4. The execution time of the first job released at time zero is $c_1[0] = 0.65$. With a budget allocation of 0.25, the job is not able to finish by the corresponding deadline, $d_{1,0} = 2$. In each reservation period from 0 to 2, Q_1 is depleted while L_1 is always more than zero.

The second job for τ_1, however, has an execution time of $c_1[1] = 0.3$ and so the budget is more than sufficient for the job to complete before the deadline $d_{1,1} = 4$. By the end of the fourth reservation period, there is no more computation remaining while there is still budget remaining. While any remaining computation at the end of

a reservation period rolls over to the next reservation period, any remaining budget does not roll over.

The third job from task τ_1 has an execution time that is just large enough to deplete the budget allocation and still small enough for the job to complete. Therefore, both Q_1 and L_1 are zero by the end of the sixth reservation period.

In classical priority-based scheduling, the utilization factor for a task set is defined in terms of task periods and WCETs as shown in (2). A set of tasks are guaranteed to meet corresponding deadlines if the utilization factor is less than some least upper bound (LUB). The utilization factor can be similarly defined for a set of budget-allocations by replacing the task period with the reservation period and replacing the WCET with the allocated budget. The fraction of time committed to a task τ_i during reservation period k is referred to as the bandwidth, $\beta_i[k] \in [0, 1)$. The bandwidth is therefore a ratio of the budget and the length of the reservation period: $\beta_i[k] = Q_i[k]/T_R$. The utilization factor for a set of budget allocations can be defined as the sum of the corresponding bandwidths. Provided that the utilization factor is less than the LUB, it can be guaranteed that all tasks will receive the respective allocated budgets.

$$U = \sum_i \beta_i[k] \leqslant LUB. \tag{7}$$

The value of the LUB depends on how tasks are scheduled. LUB is 1.0 when tasks are scheduled according to EDF or may be less when tasks are scheduled according to RM. Since the budget allocations for tasks are computed independently, a central component must determine if the set of budget allocations are feasible. If it is found that (7) is not satisfied, it is reffered to as an overload condition and appropriate action must be taken to ensure that (7) is satisfied. In PBS, overload conditions are addressed with the *compression* approach discussed in [5], where the allocated budgets are scaled back proportionally. The scaled budgets, $Q_{sat\,i}[k]$, are computed

as follows:

$$Q_{sat\,i}[k] = \begin{cases} \dfrac{Q_i[k] \cdot LUB}{U} & \text{iff } U > LUB \\ Q_i[k] & \text{otherwise} \end{cases} \tag{8}$$

where U denotes the utilization factor defined in (7) and LUB denotes the least upper bound for the scheduling algorithm. This *compression* approach allows for a fair degradation in the QOS under overload conditions as shown in Section 3.6.1.

Also, in the implementation of PBS discussed in Section 3.5, tasks are restricted to have task periods that are integer multiples of a common reservation period and task-period boundaries are required to be aligned with the reservation-period boundary. These restrictions allow the LUB to be 1.0.

3.1.2 Virtual Finishing Time

A budget allocation only guarantees that a certain amount of CPU time will be committed to a task in a reservation period and does not guarantee the time period within the reservation period when the task is allowed to run. Therefore, the latest possible finishing time (LFT) of a job is the end time of the reservation period in which the job completes.

The virtual finishing time (VFT) of a job is a measure of when that job would finish if it ran on a dedicated virtual processor whose speed is a fraction of that of the physical processor. The fraction is defined by the bandwidth, β_i, described in Section 3.1.1. LFT is a quantized form of VFT, rounding VFT up to the nearest multiple of T_R. An expression relating VFT to LFT is as follows:

$$VFT_i[j] = LFT_i[j] - \frac{Q_{left\,i}[j] \cdot T_R}{Q_i[k]}, \tag{9}$$

where $Q_{left\,i}[j]$ denotes the unused budget remaining when the job completes and $Q_i[k]$ denotes the budget allocated in that reservation period.

The VFT of a job is measured relative to the corresponding release time. A VFT greater than the deadline indicates a missed deadline and insufficient budget

allocation. A VFT less than the deadline indicates an early finish and excess budget allocation. Ideally, the VFT of a job should be equal to the job deadline. Therefore, the scheduling error for a job is defined as the difference between the VFT and the corresponding deadline. In the task model considered in this dissertation, the relative job deadline is the task period. An expression for the VFT error of a job is as follows:

$$\epsilon[j] = \left(LFT[j] - \frac{Q_{left\,i}[k] \cdot T_R}{Q_i[k]} \right) - T. \tag{10}$$

Positive values of ϵ imply better utilization at the cost of tardiness, whereas negative values of ϵ imply timeliness at the cost of poor utilization. The VFT error is often normalized by the task period to allow for comparison between tasks of different periods. VFT and related dynamics are discussed in greater detail in [6].

3.2 Prediction-based Budget Allocation

The goal of an adaptive budget allocator is to assign appropriate levels of CPU budget, $Q_i[k]$, to a task such that as many jobs as possible are able to meet corresponding deadlines while the utilization of the allocated budget is kept as high as possible. Before proceeding with the description of PBS, an ideal allocation algorithm is described. In the following discussion, values indexed by k are sampled at the beginning of the reservation period after any jobs are released. Decisions regarding budget allocation are local to each task. Therefore, to simplify the notation in this section, the subscript i representing the task index is dropped from all variables. All variables correspond to the same task except when stated otherwise.

Over a given task period, the ideal budget allocation should allow the last released job to complete before the corresponding deadline at the end of that task period. However, the ideal level of CPU budget should be completely depleted to allow for a budget utilization of 100%. Lastly, the ideal sequence of budget allocations should be evenly distributed across the reservation periods of the task period.

Let K denote the number of reservation periods in a task period. In the k^{th} reservation period, the number of reservation periods remaining until the end of the current task period can be expressed as $K-(k \mod K)$. Therefore, the ideal budget allocation can be expressed according to (11) where $L[k]$ denotes the computational load.

$$Q_{ideal}[k] = \frac{L[k]}{K-(k \mod K)} \tag{11}$$

However, the computational load, $L[k]$, is made up of the execution times of jobs that have not yet finished and therefore, is unknown. In PBS, job-execution times are modeled as a random process and the computational load is predicted accordingly. Then, the budget is allocated based on the predicted value of $L[k]$.

3.2.1 The PBS Algorithm

In the design of PBS, it is assumes that the budget allocator has access to a number of variables. The first is the number of jobs that have been released but not completed, referred to as *queue length*, $l[k]$. Also, it is assumed that the budget allocator has access to the elapsed execution time, $c_{min}[k]$, of the current job that has started running but not yet finished. The elapsed execution time is by default the minimum execution time of the currently running job. In addition, the budget allocator is assumed to have access to the execution time of recently completed jobs. Lastly, the budget allocator is able to determine the number of reservation periods remaining until the deadline of the last released job. The last released job is the one released at the beginning of the current task period and has a deadline at the end of the current task period. Based on these variables, the value of the allocated budget, $Q[k]$ is assigned as follows:

$$Q[k] = \begin{cases} \dfrac{\tilde{c}_{cond}[k] - c_{min}[k] + (l[k]-1)\tilde{c}[k]}{K-(k \mod K)} & \text{iff } l[k] > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{12}$$

The denominator is the same as in the ideal budget allocation shown in (11) but the numerator is an estimate of $L[k]$. The variables $c_{min}[k]$ and $l[k]$ are as described above. The variable $\tilde{c}[k]$ denotes an approximation of the mean execution time. The variable $\tilde{c}_{cond}[k]$ is an approximation of the conditional mean of the execution time given that the execution time is known to be larger than $c_{min}[k]$:

$$\tilde{c}_{cond}[k] \approx E(c|c > c_{min}[k]). \tag{13}$$

The mean execution time, $\tilde{c}[k]$, is approximated using the sample mean of the execution times of the N most recently completed jobs. The history length, N, is a configurable parameter satisfying $N \geq 2$. For the purposes of computing conditional means, job-execution times are assumed to have a translated exponential distribution. As shown in Section 3.6.2, this assumption is appropriate for sufficiently long jobs. Accordingly, the conditional mean, $\tilde{c}_{cond}[k]$, is approximated using (14) where $\hat{\sigma}[k]$ is the sample standard deviation of the execution times of the N most recently completed jobs. The α parameter is a user-defined parameter satisfying $\alpha \geq 1.0$. The effects of varying α on budget utilization and miss rate are discussed in Section 3.6.2.

$$\tilde{c}_{cond}[k] = \begin{cases} \tilde{c}[k] + (\alpha - 1)\hat{\sigma}[k] & \text{for } (\tilde{c}[k] - \hat{\sigma}[k]) \geq c_{min}[k] \\ c_{min}[k] + \alpha\hat{\sigma}[k] & \text{otherwise} \end{cases} \tag{14}$$

3.2.2 How PBS Works

To understand how budget is allocated using (14), four different cases are considered. The first case is a reservation period that is also the beginning of a task period and there are no overruns from previous task periods. In this case, $l[k] = 1$, $c_{min}[k] = 0$, and the budget allocation in this case is as follows:

$$Q[k] = \frac{\tilde{c}_{cond}[k]}{K}. \tag{15}$$

The second case is a reservation period that is not the beginning of a task period and there are no overruns from previous task periods. In this case, $l[k] = 1$,

$c_{min}[k] > 0$, and the budget is allocated as follows:

$$Q[k] = \frac{\tilde{c}_{cond}[k] - c_{min}[k]}{K - (k \mod K)}. \tag{16}$$

The third case is a reservation period that may or may not be the beginning of a task period and there has been overruns by previous jobs. In this case, $l[k] > 1$, $c_{min}[k] \geq 0$, and the budget allocation is calculated in the full form as

$$Q[k] = \frac{\tilde{c}_{cond}[k] - c_{min}[k] + (l[k] - 1)\tilde{c}[k]}{K - (k \mod K)}. \tag{17}$$

The last case is when a job completes prior to the last reservation period before the corresponding deadline. In that case, $l[k] = 0$ and no budget is allocated for those last reservation periods.

3.3 Queue Stability

A queue is stable if the number of elements in the queue remains bounded. In this section, it is proved that budget allocation according to the PBS algorithm allows the job queue of a task to remain stable.

Jobs can only arrive at task-period boundaries and exactly one job arrives at each task-period boundary. It can be argued that the queue length is stable if for initial queue lengths sufficiently large, the queue length decreases on average as expressed in (18). The variable $l_T[j]$ is the queue length at the beginning of j^{th} task period. Since there are K reservation periods in a task period, $l_T[j]$ is related to $l[k]$ based on $l_T[j] = l[K \cdot j]$.

$$\mathbf{E}(l_T[j+1] \mid l_T[j] = l_1) < l_1 \ \forall l_1 \geq N \tag{18}$$

Theorem 1 which will be proved in Section 3.3.5 establishes that *when the job-execution times are independently and identically distributed (i.i.d.) and tasks are allocated budget according to (12), sufficient budget is allocated by the end of the last*

reservation period of a task period such that (18) is satisfied. The condition of i.i.d. execution times is a sufficient and not necessary condition.

The remainder of this section is organized as follows. In Section 3.3.2, expressions are derived to relate the total budget allocated over a task period ($Q_{total}[j]$), the computational load ($L_l[j]$), and the estimated mean execution time ($\tilde{c}[k]$), to the execution times of jobs that are completed and queued. In Section 3.3.3, the expected value of the queue length, $l_T[j+1]$, in the $(j+1)^{th}$ task period conditioned on the value of previous queue-lengths is reduced to a summation of conditional-probability terms. In Section 3.3.4, the conditional-probability terms are expressed in terms of job-execution times using the expressions from Section 3.3.2. Also, a number of inequalities are defined on these probability terms. Finally, in Section 3.3.5, these inequalities are used to prove queue stability as described above. In Section 3.3.6, a discussion is presented on the implications of queue stability.

3.3.1 Notation

Before proceeding, the following notation is introduced for convenience. Let $k_{first} = (j \cdot K)$ denote the index of the first reservation period, and let $k_{last} = (j \cdot K + K - 1)$ denote the index of the last reservation period of the j^{th} task period.

Let the summation of the execution times of a specific sequence of jobs be denoted as

$$\overset{j_2}{\underset{j_1}{\mathrm{C}}} \triangleq \begin{cases} \sum_{j=j_1}^{j_2} c[j] & \text{if } j_2 \geq j_1 \\ 0 & \text{otherwise} \end{cases}. \tag{19}$$

Since job-execution times are i.i.d., the specific indices of jobs are only useful in establishing independence. Once independence is established, it is possible to represent the sums of execution times in more compact form as a function of only the number of jobs. Let the summation of the execution times of an arbitrary and

independent set of jobs be denoted as

$$C_A \triangleq \sum_{j=1}^{A} c[j] \tag{20}$$

where A in (20) represents the number of jobs in the summation.

3.3.2 Load, Budget, and Estimated Mean

Let the computational load, $L_l[j]$, of the first l jobs in the job queue be defined as the dedicated CPU time needed by the task at the beginning of the j^{th} task period to complete those jobs. In the j^{th} task period, the last job in the queue (also the most recent job) is J_j. Given that the queue length is $l_T[j] = l[k_{first}]$ and that jobs complete in order, the first queued job is $J_{j-l[k_{first}]+1}$. Similarly, the l^{th} job in the queue is $J_{j-l[k_{first}]+l}$. Furthermore, the first queued job has already run in one or more previous reservation periods for some amount of time $c_{min}[k_{first}]$. Therefore, $c_{min}[k_{first}]$ is no longer a part of the current load and $L_l[j]$ can be expressed as

$$L_l[j] = \mathrm{C}_{j-l[k_{first}]+1}^{j-l[k_{first}]+l} - c_{min}[k_{first}]. \tag{21}$$

Next, consider $Q_{total}[j]$, the total budget allocated during the j^{th} task period. This term can be separated into two parts: the budget allocated *until* the last reservation period and the budget allocated *in* the last reservation period.

$$Q_{total}[j] = \sum_{k=k_{first}}^{k_{last}-1} Q[k] + Q[k_{last}]. \tag{22}$$

The budget allocated until the last reservation period consists of the execution times of jobs completed by the start of the last reservation period and any time, $c_{min}[k_{last}]$, spent executing an additional job that did not finish. Furthermore, the first queued job will have already run some amount of time, $c_{min}[k_{first}]$, in previous reservation periods. This time is not a part of the consumed budget in the j^{th} task period. Therefore, the budget allocated in the first $(K-1)$ reservation periods can be expressed

as

$$\sum_{k=k_{first}}^{k_{last}-1} Q[k] = \overset{j-l[k_{last}]}{\underset{j-l[k_{first}]+1}{\mathrm{C}}} + c_{min}[k_{last}] - c_{min}[k_{first}]. \tag{23}$$

The budget allocated in a reservation period is defined by (12). In the last reservation period of a task period, the denominator in (12) becomes one and the allocated budget is computed as

$$Q[k_{last}] = \tilde{c}_{cond}[k_{last}] - c_{min}[k_{last}] + (l[k_{last}] - 1)\tilde{c}[k_{last}]. \tag{24}$$

Since no jobs arrive between task-period boundaries, $l[k_{first}] \geq l[k_{last}]$. Substituting (23) and (24) into (22), the total budget can be expressed as

$$Q_{total}[j] = \overset{j-l[k_{last}]}{\underset{j-l[k_{first}]+1}{\mathrm{C}}} - c_{min}[k_{first}] + \tilde{c}_{cond}[k_{last}] + (l[k_{last}] - 1)\tilde{c}[k_{last}]. \tag{25}$$

As a lower bound on (25), let $\hat{Q}_{total}[j]$ be defined according to (26).

$$\hat{Q}_{total}[j] = \overset{j-l[k_{last}]}{\underset{j-l[k_{first}]+1}{\mathrm{C}}} - c_{min}[k_{first}] + l[k_{last}]\tilde{c}[k_{last}]. \tag{26}$$

Since by definition $\tilde{c}_{cond}[k_{last}] \geq \tilde{c}[k_{last}]$, $\hat{Q}_{total}[j]$ is a lower bound on the total allocated budget, $Q_{total}[j]$.

Lastly, consider $\tilde{c}[k]$, the estimated mean execution time. This estimate is computed as the sample mean of the execution times of the N most recently completed jobs where N is the history length defined in Section 3.2.1. The first queued job is $J_{j-l[k]+1}$. Therefore, the last completed job is $J_{j-l[k]}$ and the estimate of the mean execution time can be expressed as

$$\tilde{c}[k] = \frac{1}{N} \overset{j-l[k]}{\underset{j-l[k]-N+1}{\mathrm{C}}}. \tag{27}$$

3.3.3 Expected Queue Length

To prove that (18) holds, an expression is needed for the expected value of queue length, $l_T[j+1]$, at the start of the $(j+1)^{th}$ task period conditioned on the value of the queue length, $l_T[j]$, at the start of the j^{th} task period. As a stepping stone, an

expression is derived for the expected queue length conditioned further on the queue length, $l[k_{last}]$, in the last reservation-period of the j^{th} task period. For convenience, let $\Phi(l_1, l_{last})$ denote the conditioning event that $(l_T[j] = l_1)$ and $(l[k_{last}] = l_{last})$. By definition of expected value,

$$\begin{aligned} E(l_T[j+1] \mid \Phi(l_1, l_{last})) &= \sum_{l=1}^{l_1+1} l \cdot \Pr(l_T[j+1] = l \mid \Phi(l_1, l_{last})) \\ &= \sum_{i=1}^{l_1+1} \Pr(l_T[j+1] \geq i \mid \Phi(l_1, l_{last})). \end{aligned} \tag{28}$$

The terms in the summation in (28) are equivalent to the probability that at least i jobs remain in the queue at the beginning of the $(j+1)^{th}$ task period. Since exactly one job arrives at the beginning of each task period, the probability is one that $l_T[j+1] \geq 1$. Similarly, the probability is zero that $l_T[j+1] > (l_{last}+1)$. Eliminating zero-valued terms, the expression can be reduced as shown in (29).

$$\begin{aligned} E(l_T[j+1] \mid \Phi(l_1, l_{last})) &= \sum_{i=1}^{l_1+1} \Pr(l_T[j+1] \geq i \mid \Phi(l_1, l_{last})) \\ &= 1 + \sum_{i=2}^{l_{last}+1} p(l_1, l_{last}, i). \end{aligned} \tag{29}$$

The probability term $p(l_1, l_{last}, i)$ is defined according to (30) over the domain

$$\{l_1, l_{last}, i \in \mathbb{Z}^+ : l_1 \geq l_{last} \geq (i-1) \geq 1\}.$$

$$p(l_1, l_{last}, i) = \Pr(l_T[j+1] \geq i \mid \Phi(l_1, l_{last})). \tag{30}$$

Given the corresponding domain, $p(l_1, l_{last}, i)$ is equivalent to the probability that at least $(i-1)$ jobs remain queued at the end of the j^{th} task period. Therefore, $p(l_1, l_{last}, i)$ is equivalent to the probability that no more than $(l_1 - i + 1)$ jobs complete in that task period. This is the probability that the total budget, $Q_{total}[j]$, allocated during the j^{th} task period is less than the computational load, $L_{(l_1-i+2)}[j]$, of the first $(l_1 - i + 2)$ queued jobs. Note, that the event $((Q_{total}[j]) < L_{(l_1-i+2)}[j])$ allows

the $(l_1 - i + 2)^{th}$ queued job to start but not complete. The modified expression for $p(l_1, l_{last}, i)$ is as follows:

$$p(l_1, l_{last}, i) = \Pr(Q_{total}[j] < L_{(l_1-i+2)}[j] \mid \Phi(l_1, l_{last})). \tag{31}$$

Given that $\hat{Q}_{total}[j]$ is a lower bound on $Q_{total}[j]$, let $\hat{p}(l_1, l_{last}, i)$ denote an upper bound on $p(l_1, l_{last}, i)$ defined as follows:

$$\hat{p}(l_1, l_{last}, i) = \Pr(\hat{Q}_{total}[j] < L_{(l_1-i+2)}[j] \mid \Phi(l_1, l_{last}))\text{..} \tag{32}$$

3.3.4 Conditional Probability of Minimum Queue Length

Replacing $L_{(l_1-i+2)}[j]$ and $\hat{Q}_{total}[j]$ in (32) with the corresponding expressions in (21) and (26), respectively, $\hat{p}(l_1, l_{last}, i)$ can be defined in terms of job execution times. Removing common terms from the two sides in (33), the expression can be simplified further. Finally, replacing $\tilde{c}[k_{last}]$ in (34) with the corresponding expression in (27) yields (35).

$$\hat{p}(l_1, l_{last}, i) = \Pr\left(\left(\underset{j-l_1+1}{\overset{j-l_{last}}{\mathrm{C}}} + l_{last}\ \tilde{c}[k_{last}]\right) < \underset{j-l_1+1}{\overset{j-i+2}{\mathrm{C}}}\right) \tag{33}$$

$$= \Pr\left(\tilde{c}[k_{last}] < \frac{1}{l_{last}} \underset{j-l_{last}+1}{\overset{j-i+2}{\mathrm{C}}}\right) \tag{34}$$

$$= \Pr\left(\frac{1}{N} \underset{j-l_{last}-N+1}{\overset{j-l_{last}}{\mathrm{C}}} < \frac{1}{l_{last}} \underset{j-l_{last}+1}{\overset{j-i+2}{\mathrm{C}}}\right) \tag{35}$$

The two sides of the inequality in (35) are made up of execution times of different jobs and therefore, the two sides are independent. Disregarding the job indices and considering only the number of jobs allows $\hat{p}(l_1, l_{last}, i)$ to be expressed as shown in (36).

$$\hat{p}(l_1, l_{last}, i) = \Pr\left(\frac{1}{N} C_N < \frac{1}{l_{last}} C_{(l_{last}-i+2)}\right) \tag{36}$$

Based on Equation (36), a number of inequalities are defined on $\hat{p}(l_1, l_{last}, i)$ for use in the proof of Theorem 1 below.

$$\hat{p}(l_1, l_{last}, (i+1)) < \hat{p}(l_1, l_{last}, i) \tag{37a}$$

$$\hat{p}(l_1, (l_{last}+1), (i+1)) < \hat{p}(l_1, l_{last}, i) \tag{37b}$$

$$\hat{p}(l_1, l_{last}, i) < 1.0 \tag{37c}$$

$$\hat{p}(l_1, N, 2) = 0.5 \tag{37d}$$

$$\forall\ l_{last} \geq N \text{ and } i \geq (l_{last} - N + 2)$$

$$\hat{p}(l_1, l_{last}, i) \leq 0.5 \tag{37e}$$

Plugging the appropriate values of i into (36), it can be seen that (37a) is true.

$$\Pr\left(\frac{1}{N}C_N < \frac{1}{l_{last}}C_{(l_{last}-i+1)}\right) < \Pr\left(\frac{1}{N}C_N < \frac{1}{l_{last}}C_{(l_{last}-i+2)}\right)$$

Similarly, Inequality (37b) can be confirmed by setting the second and third parameters of (36) to $(l_{last}+1)$ and $(i+1)$, respectively.

$$\Pr\left(\frac{1}{N}C_N < \frac{1}{(l_{last}+1)}C_{(l_{last}-i+2)}\right) < \Pr\left(\frac{1}{N}C_N < \frac{1}{l_{last}}C_{(l_{last}-i+2)}\right)$$

As mentioned at the beginning of Section 3.3, job-execution times are i.i.d. Therefore, when the third parameter of (36) is set to two, the left and right-hand side of the inequality have the same mean and so $\hat{p}(l_1, l_{last}, 2)$ is strictly less than 1.0.

$$\hat{p}(l_1, l_{last}, 2) = \Pr\left(\frac{1}{N}C_N < \frac{1}{l_{last}}C_{l_{last}}\right) < 1.0$$

Furthermore, based on the domain of $\hat{p}(l_1, l_{last}, i)$ shown in (30), $i \geq 2$. Taking into account Inequality (37a), it can be seen that (37c) is true.

$$\hat{p}(l_1, l_{last}, i) \leq \hat{p}(l_1, l_{last}, 2) < 1.0$$

For $l_{last} = N$ and $i = 2$, the two sides of the inequality in (36) are i.i.d., which confirms (37d).

$$\hat{p}(l_1, N, 2) = \Pr\left(\frac{1}{N}C_N < \frac{1}{N}C_N\right) = 0.5$$

Finally, the derivation of (37e) is shown in the steps below. Applying Inequality (37b) and (37a), respectively, to $\hat{p}(l_1, l_{last}, i)$ yields (38b). Finally applying (37d) to (38b) shows that (37e) is true.

$$\begin{aligned}
&\forall\ l_{last} \geq N \text{ and } i \geq (l_{last} - N + 2) \\
&\hat{p}(l_1, l_{last}, i) \\
&\leq \hat{p}(l_1, N, i - (l_{last} - N)) && \text{(38a)} \\
&\leq \hat{p}(l_1, N, 2) && \text{(38b)} \\
&= 0.5
\end{aligned}$$

3.3.5 Queue Stability

Based on Equation (29) and inequalities (37a) through (37e), it can be shown now that budget allocated according to (12) results in stable job queues as expressed in Theorem 1 below.

Theorem 1. *If job execution times are i.i.d. and CPU budget is allocated according to (12), then queue lengths satisfy (39).*

$$E\left(l_T[j+1] \mid l_T[j] = l_1, l[k_{last}] = l_{last}\right) < l_1 \quad \forall l_1 \geq N, l_{last} \leq l_1 \tag{39}$$

Proof: The proof of Theorem 1 can be divided into four cases that cover all valid values of $l_{last} \leq l_1$ and $l_1 \geq N$. The proof is trivial for the case when $l_{last} < (l_1 - 1)$ because no more than one job can arrive at a task-period boundary and so $l_T[j+1]$ is less than or equal to $(l[k_{last}]+1)$. This leaves the case when $l_{last} = (l_1 - 1)$ and the case when $l_{last} = l_1$. The latter is separated further into two cases for convenience. The proof of Theorem 1 for each of these three remaining cases is presented below. For each case, the proof begins with the expression for $E(l_T[j+1] \mid \Phi(l_1, l_{last}))$ presented in (29).

Case 1: $l_{last} = l_1 - 1$

$$E(l_T[j+1] \mid \Phi(l_1, l_1 - 1)) = 1 + \sum_{i=2}^{l_1} p(l_1, l_1 - 1, i)$$

$$\leq 1 + \sum_{i=2}^{l_1} \hat{p}(l_1, l_1 - 1, i) \tag{40a}$$

$$< 1 + (l_1 - 1) \tag{40b}$$

Applying (37c) to (40a) yields (40b), prooving Theorem 1 for Case 1.

Case 2: $l_{last} = l_1 = N$

$$E(l_T[j+1] \mid \Phi(N, N)) = 1 + \sum_{i=2}^{N+1} p(N, N, i)$$

$$\leq 1 + \sum_{i=2}^{N+1} \hat{p}(N, N, i) \tag{41a}$$

$$< 1 + \sum_{i=2}^{N+1} \hat{p}(N, N, 2) \tag{41b}$$

$$< 1 + \sum_{i=2}^{N+1} 0.5 \tag{41c}$$

Applying (37a) and (37d) to (41a) yields (41b) and then (41c), respectively. Since $N \geq 2$, this step proves Theorem 1 for Case 2.

Case 3: $l_{last} = l_1 > N$

$$E(l_T[j+1] \mid \Phi(l_1, l_1)) = 1 + \sum_{i=2}^{N+1} p(l_1, l_1, i)$$

$$\leq 1 + \sum_{i=2}^{l_1+1} \hat{p}(l_1, l_1, i)$$

$$= 1 + \sum_{i=2}^{l_1-N+1} \hat{p}(l_1, l_1, i) + \sum_{i=l_1-N+2}^{l_1+1} \hat{p}(l_1, l_1, i) \tag{42}$$

$$< 1 + (l_1 - N) + (0.5N) \tag{43}$$

Applying (37c) and (37e) to the left and right summations in (42) yields (43). Since $N \geq 2$, this last step proves Theorem 1 for Case 3. □

In Theorem 1, $l[k_{last}] = l_{last}$ is required to be less than or equal to $l_T[j] = l_1$. Since no jobs can arrive in the middle of a task period, this constraint encompasses all valid values of l_{last} and therefore, (39) holds irrespective of the value of l_{last}. Applying the law of total probability to the left hand side of Inequality (39) over all valid values of l_{last}, it can be proved that (18) is satisfied.

3.3.6 Further Discussion

Given the periodic and sequential nature of jobs, bounded queue lengths also impose bounds on job-finishing times and scheduling errors as described in Section 3.1.2. Furthermore, since budget allocations are based on queue length and estimates of the mean execution time, bounded queue lengths and execution times also impose bounds on the allocated budget.

While it is assumed in the proof of Theorem 1 that all job-execution times are i.i.d., the proof only requires the execution times of queued jobs and recently completed jobs to be i.i.d. Lastly, i.i.d. execution times are a sufficient and not necessary condition for queue stability. It is shown in Section 3.6 that while execution times measured from actual workloads are not i.i.d., the PBS algorithm is still stable and performs well.

3.4 Simulation-Based Comparison

To demonstrate the advantage of asynchronous budget adaptation at reservation period boundaries, the performance of PBS is compared to that of a synchronous policy called Finishing-Time Update (FTU). In the FTU policy, the allocated budget is computed on job completion as follows:

$$Q[k] = \tilde{c}_{cond}[k] + (l[k])\tilde{c}[k]. \tag{44}$$

For simplicity, the task period is set to equal the reservation period ($K = 1$). Taking into account that $c_{min}[k] = 0$ on job completion, the FTU policy is identical to PBS except that the budget adaptation is performed on job completion. Also, the FTU policy is similar to the Stochastic Dead Beat approach described in [5] when deadlines are met and $l[k] = 0$.

The performance of PBS and FTU is evaluated using the MATLAB-based simulator described in [8]. The simulations involve a synthetic workload where execution times are generated as a noisy pulse. The job-execution times are plotted against job release times in Figure 5. This load is meant to test the responsiveness of PBS and FTU to a sudden and severe change in the average job-execution time.

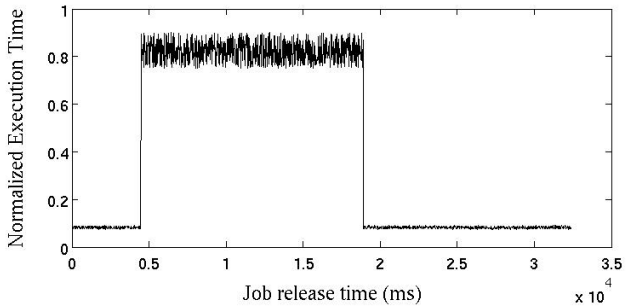


Figure 5: Job execution times from a synthetic workload.

The normalized VFT error for the two policies are shown in Figure 6. It is clear from the figure that the PBS algorithm incurs a smaller scheduling error for the first long job and recovers faster from the step change in execution time than the FTU algorithm. Similar results are anticipated for any budget adaptation policy where the adaptation is performed on job completion because such a policy is not able to adapt to the step change until the first long job completes.

3.5 Software Architecture for an Asynchronous Budget Mechanism

Implementation of adaptive reservation-based schedulers has been explored on a number of different operating systems. Most implementations presented in the reviewed

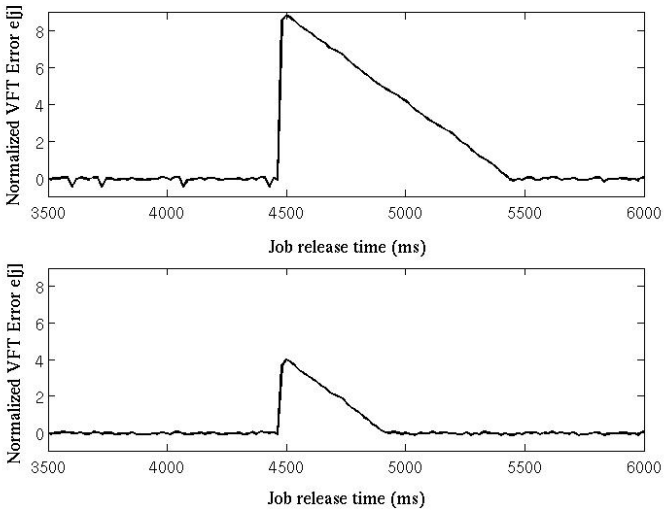


Figure 6: VFT error for the FTU policy (top) and PBS policy (bottom) with the synthetic workload shown in Figure 5.

literature are based on the Linux kernel. Although Linux is not considered a real-time kernel, the response time of Linux can be improved through a number of different mechanisms as shown in [35]. Along the same lines, `PREEMPT_RT` is a set of patches for the Linux kernel meant to make the Linux kernel preemptible and more responsive[52]. A key component of adaptive budget allocation is execution-time monitoring. A mechanism for accounting execution times in Linux is presented in [13]. The approach used in the prototype presented in this section is similar but does not involve patching the kernel.

There are a number of previous and current projects on adaptive reservation-based scheduling in Linux. OCERA [28] is one of the earlier projects and implements the policy described in [5]. AQuoSA [59] is a port of OCERA to the 2.6 version of the Linux kernel. A current project aims to integrate the CBS scheduling algorithm into the mainstream Linux kernel as a new scheduling class called `SCHED_EDF` [32].

These implementations [28][59][5] are appropriate for approaches where budget adaptations are performed on job completion. However, in the PBS algorithm, budget adaptations are performed at reservation-period boundaries often in the middle of

jobs. Since budget adaptations are asynchronous, a separate execution context is needed to perform budget adaptations that is different from the execution context in which the jobs are run. To this end, the use of a daemon process is proposed called the "Allocator" task. The overall structure of the system is presented in Figure 7. The complete system consists of three parts: 1) the PBS kernel module to implement the necessary functionality in kernel space, 2) a daemon process called Allocator to perform the asynchronous budget allocation, and 3) a small library used by SRT tasks to interact with the rest of the system.

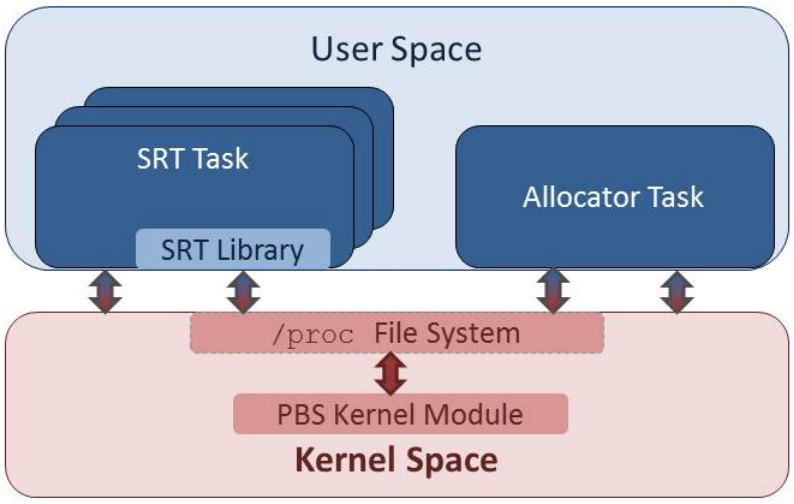


Figure 7: Overall structure of the asynchronous-budget mechanism.

3.5.1 SRT Tasks

An SRT task is assumed to have a basic structure as shown in Figure 8. After any application-specific initialization operations, an SRT task is registered with the PBS kernel module and parameters such as task period, history length and initial budget are configured. Then, the task enters the main loop where each iteration of the loop corresponds to a job. Each loop iteration begins by notifying the PBS module that the previous job has ended and that the next job is about to begin.

```
...
/*application-specific initialization*/
...
/*register the task with the PBS module as an SRT
task*/
handle = open("/proc/ ...
...
/*configure the task period, history length, budget allo-
cation*/
ioctl(handle, ...
...
while  there are jobs remaining do
    /* Notify PBS module of job boundary */
    read(handle, ...
    ...
    /* job-specific computation */
    ...
end
/* Notify PBS module of end of task*/
close(handle);
/*application-specific cleanup*/
...
```

Figure 8: Basic template of an SRT task.

Timing for SRT tasks is handled by the kernel module. The module is notified of the task period during the initialization stage. Based on the task period and the notifications on job completion, the module keeps track of the number of queued jobs. When there are no queued jobs and the PBS module is notified that the current job is complete, the corresponding task is blocked until the next task period. If the job that just completed missed the corresponding deadline, the task is allowed to continue without blocking to the next queued job.

In the prototyped system, any throttling related to budget enforcement is transparent to the SRT task. The task is simply preempted when the allocated budget is depleted and rescheduled in the following reservation period when the budget is replenished. When the last job ends, the SRT task exits the main loop and the PBS module is notified that there are no remaining jobs.

The SRT tasks interact with the PBS module and the Allocator task through a

special file in the Linux `/proc` file system. Initialization operations consist of calls to `open` and `ioctl`. The "end of job" notifications involves a call to `read`. The "end of task" notification involves a call to `close`.

3.5.2 The Kernel Module

Any scheduling-related functionality that needs to be in kernel space is implemented in the PBS kernel module. This functionality includes timing, budget enforcement, and tracking various parameters such as job-queue lengths, process run-times, and current and past job-execution times.

All SRT tasks are registered with the kernel module during their initialization phase. During registration, the necessary data structures are allocated in the kernel. Also, the module associates a `preempt_notifier` with newly registered tasks. A `preempt_notifier` is a mechanism in the Linux kernel that allows callback functions to be called whenever tasks are preempted or scheduled to run. These notifiers, in combination with the `sched_clock` function, allow for accurate measurement of job execution times. On x86 machines, `sched_clock` is based on `TSC`.

For timing and budget enforcement, High-resolution timers (HR-timer) are used. A central HR-timer called `sp_timer` is fired at the beginning of every reservation period. When this timer fires, a sorted queue of timing events is used to determine if blocked SRT tasks should be woken and if corresponding job-queue lengths should be incremented. This timing queue consists of the start times of the next task-period boundary of all running SRT tasks and is updated at each task-period boundary. Also, when the `sp_timer` fires, all budget allocations are replenished and any throttled task is unblocked.

An additional HR-timer is associated with every SRT task for budget enforcement. Whenever a task is scheduled to run and the `preempt_notifier` callback function is called, this budget-enforcement timer is setup to fire when the allocated budget is

depleted. If this timer fires, the task is put to sleep and a special `THROTTLED` flag is set for the task. On the other hand, if the task goes to sleep and the corresponding `preempt_notifier` callback function is called, the budget-enforcement timer is disabled and any necessary accounting is performed. This dynamic-timer-based approach to budget enforcement is more accurate than periodic-tick-based accounting and enforcement.

3.5.3 The Allocator Daemon

The Allocator daemon, referred to as the Allocator, is a user-level process where SRT budget allocations are computed. The Allocator daemon is activated periodically at the beginning of every reservation period. In order to ensure that budget updates are done on time, the Allocator is scheduled as an HRT task with a fixed budget and at a higher priority than SRT tasks.

The structure of the Allocator is similar to that of SRT tasks. Initialization involves opening a specific file in the `/proc` file system and configuring the reservation-period length through `ioctl` calls to the file. Once the initialization is complete, the Allocator enters a main loop. An iteration of this loop is executed every reservation period. In each iteration, the allocator is blocked until the start of the next reservation period by calling `read` on the file described earlier. On returning from the call, the Allocator goes through the data structure associated with each SRT task in the system and the budget allocated to each task is recomputed. When the allocator goes to the next iteration and calls `read`, control returns to kernel space. In the kernel, the budget allocations are checked and scaled back to ensure schedulability as described in Section 3.1.1. Then, the allocator is put to sleep until the next reservation-period boundary.

Given the nature of the system, a relatively large amount of information has to be shared between the Allocator and the kernel module. Copying data between kernel

space and user space can result in a large overhead. Furthermore, this overhead would be incurred on each activation of the allocator at each reservation-period boundary. In order eliminate this overhead, memory mapping is used.

The `/proc` file used for the Allocator daemon exports an `mmap` function that allows two specific regions of Kernel memory to be mapped into the Allocator address space. The first region is a read-only region that contains data used to compute the budget allocations. The second region is a writable array that is used to assign the SRT tasks the respective budgets. Having separate regions with controlled access and performing the schedulability test in kernel space allows the overall system to be more tolerant to any bugs in the Allocator code.

Given that budget adaptations have to be performed asynchronously with respect to the SRT task executions, there are two other possible implementation approaches. The first is to compute budget allocations in kernel space. The other alternative is to use UNIX signals and signal handlers to perform budget adaptations in the context of the SRT tasks.

The Allocator daemon approach, however, has a number of advantages. Development in user-level is simpler and more flexible than kernel-level development. Another advantage is that user-level applications can use floating-point and SIMD hardware which is not simple or efficient in the kernel. Furthermore, memory mapping can only be done at a granularity of memory pages. The memory-mapping technique may not be feasible if two additional regions have to be mapped for every SRT task. Lastly, signaling every SRT task in every reservation period can introduce a large amount of overhead. This overhead is avoided by the batched computation of budget allocations in the Allocator daemon.

3.6 Experiments and Results

The PBS algorithm and the asynchronous-budget-allocation mechanism were tested on the x86 Linux platform described in Table 1. The kernel was configured to disable `cgroup`-based scheduling and to be non-SMP because the system has been designed only for a single core. For experimental repeatability and consistency, the Linux scaling governor was set to `userspace` and the scaling frequency was set to a constant of 2.67GHz for all the experiments.

Table 1: Test Platform Configuration.

Processor	Intel Core 2 Extreme
Clock Speed	2.67GHz
L1 D-cache	32kB
L1 I-cache	32kB
L2 Cache	6144kB
Operating System	RHEL 6 Server
Kernel	Linux 3.1.1

While this specific fixed configuration was used for the purpose of formal experiments, similar experiments were also performed on an AMD Turion-based laptop with an Ubuntu operating system. The standard Ubuntu kernel was used in those experiments and the results obtained were similar to what is presented in this section.

For comparison with the PBS algorithm, similar experiments were performed with a static allocation policy. In the static policy, the budget allocated to a task in each reservation period is kept constant over the duration of the task's execution. The system described in Section 3.5 is still used for the static-budget-allocation policy except that the Allocator was modified to keep the allocated budget constant.

3.6.1 Performance Under Overload

The first set of experiments are based on a synthetic workload. The results from this experiment demonstrate the fair and graceful degradation in the QoS of PBS-managed tasks under overload conditions. The results further show the way a set of PBS-managed tasks recover from an overload condition.

The test application follows the template described in Section 3.5.1. The application consists of an outer loop to iterate through jobs and an inner loop to represent the computation in each job. The number of iterations of the inner loop is used to control the execution times of the jobs. The job execution times are generated as a square wave with added noise. The noise is generated using a linear congruential generator and approximates a uniform distribution. Parameters such as the period, offset, duty cycle, amplitude and noise magnitude are all configured using command-line arguments.

The total workload in the square-wave experiments consists of two running instances of the test application just described. The two tasks have task periods of 40ms and 60ms, respectively. The normalized execution times for the two tasks are shown in Figure 9. The low parts of the square wave are at 10% of the task period, whereas the high parts of the square waves are at 60% and 40% of the task periods, respectively. The added noise is 20% of the nominal value. The period, duty cycle, and phase offset of the square waves are configured so that the system is overloaded for a short interval before shifting the load from one task to the other.

With the workload described above, the α parameter and the history length N were set to 2.0 and 40, respectively. This experiment was repeated 30 times. The observed behavior was found to be consistent across the repetitions.

The normalized-VFT error signals from one of the repetitions are shown in Figure 10. The plots show that the scheduling error for both tasks recover rapidly once the overload condition ends. Furthermore, despite the differences in the two tasks

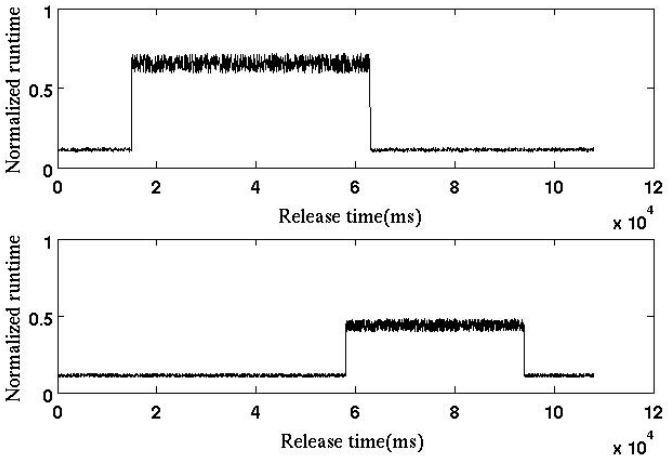


Figure 9: Computation times of two tasks in the synthetic workload.

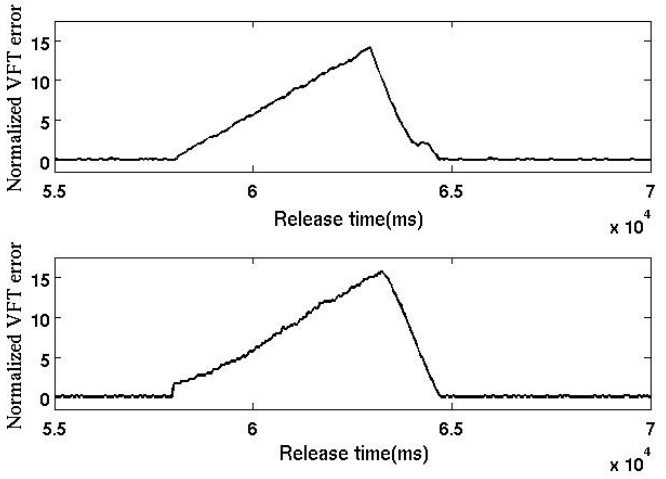


Figure 10: VFT errors for the two tasks during the overload time period for the synthetic workload shown in Figure 9.

in terms of task-period and average normalized execution time, the normalized VFT error for the two tasks increase almost evenly during the overload condition and remain roughly the same during the recovery. Lastly, the miss rates were found to be 24% for both tasks, which is inherently not possible with a static budget allocation.

3.6.2 MPEG4 Video Decoding Workload

The next set of experiments is based on MPEG4 video decoding. The test application is similar to the one described in Section 3.6.1 except that the jobs involve decoding

MPEG4 video frames. The application uses the ffmpeg library for demuxing mp4 files and decoding the frames.

During the initialization phase, all the video frames are read from the input file into memory. Any memory allocated to the application during this phase is locked to prevent paging. Each iteration of the job loop decodes a single video frame. For the purpose of these experiments, the decoded video frames are overwritten and not sent to a frame buffer.

The scheduler and test applications were tested with two separate MPEG4 videos that are referred to as v_1 and v_2, respectively. Video v_1 consists of over 6000 frames and the decoding task τ_1 is assigned a task period of 40ms. Video v_2 consists of over 2000 frames and the decoding task τ_2 is assigned a task period of 30ms. By nature of the content, the frames in v_2 are more compute-intensive to decode than the ones in v_1.

In Figure 11, two histograms are shown for the job-execution times of τ_1 and τ_2, respectively. The vertical dark lines indicate the positions of the respective means, while the vertical dashed lines are placed one standard deviation to the right of the respective means. The first key feature to note is the wide variation in execution times and the long tails in the distributions. The long tail suggests that an exponential distribution is appropriate to model the conditional distribution of the execution times of long jobs.

In Figure 12, the autocorrelation functions (ACF) of the job-execution times of τ_1 and τ_2 are shown. As mentioned in Section 3.3.6, job-execution times are not i.i.d. as assumed for the proof of queue stability. However, the i.i.d. requirement in the proof is conservative: that execution time of successive jobs should be i.i.d is a sufficient condition and not a necessary one. Based on the experimental results, it is shown that a stable queue length is still maintained with the PBS algorithm.

In the next set of experiments, the α parameter of the PBS algorithm was varied

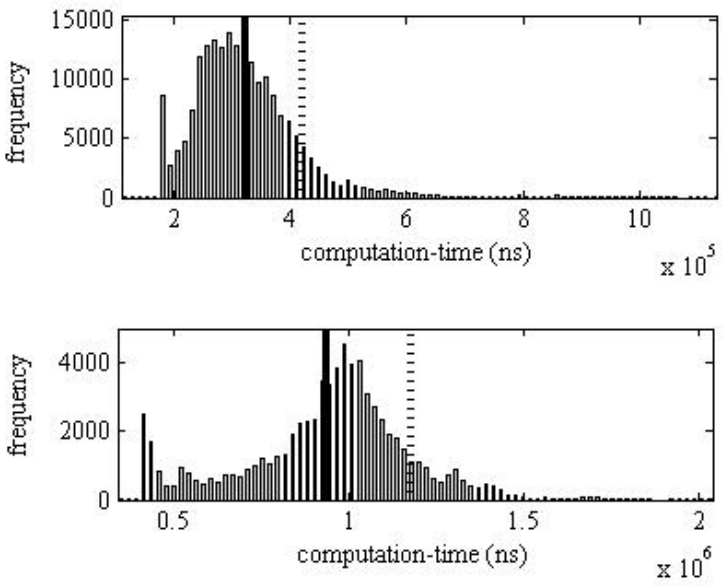


Figure 11: Histogram of job-execution times of τ_1 (top) and τ_2 (bottom), respectively.

and the scheduler was tested with the two decoding tasks described above. The resulting performance was measured in terms of average deadline miss-rate and root mean square (RMS) scheduling error. Similar experiments were performed with a static budget allocation. Instead of varying α in these experiments, the statically-allocated budget was varied. For each allocation policy, parameter value, and workload, the experiment was repeated 30 times. The performance was found to be consistent across repetitions. The presented results are averages taken across the repetitions.

As expected, different values of the α parameter in the PBS algorithm result in different miss rates. Higher values of α result in higher budget allocations on average and lower miss rates. A similar trend is seen in the results from experiments involving a static budget allocation. The higher the allocated budget, the lower the miss-rate.

Just as higher budget allocations can result in lower miss rates, it can also result in higher RMS VFT errors and lower budget utilization. Picking the appropriate value of α, or the budget in the static case, depends on the appropriate level of trade-off between miss-rate and budget utilization. This trade-off between miss-rate and

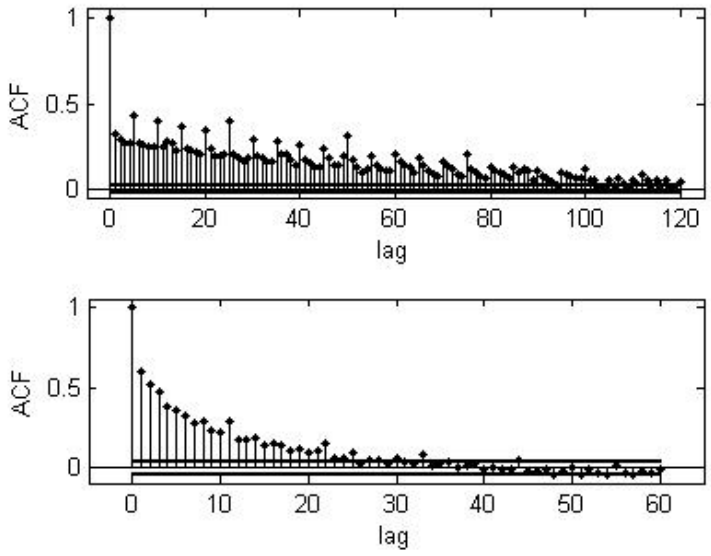


Figure 12: ACF of job-execution times of τ_1 (top) and τ_2 (bottom), respectively.

budget utilization is shown in Figure 13 and Figure 14 for the PBS-based budget allocation and static budget allocation, respectively. For proper comparison, the range of miss rates in the experiments involving the static policy had to be similar to the range of miss rates in the experiments involving PBS. Therefore, values of budget allocated in the static scheme had to be tuned. This tuning was done off-line using the actual distribution of job-execution times and trial and error.

While the ideal performance of zero miss-rate and 100% budget utilization is not possible without a priori knowledge of execution times, adaptive budget allocation based on PBS still performs closer to the ideal case than a tuned static budget allocation. For a given task and miss-rate, budget allocation based on PBS results in a lower RMS VFT-error than that resulting from a static allocation.

Furthermore, in the case of the PBS algorithm, it is possible to continue to lower RMS VFT error by allowing for a higher miss-rate. In the case of static allocation, however, the RMS VFT error reaches some minimum for some miss-rate and then begins to increase with miss-rate. This increase in RMS VFT error may be due

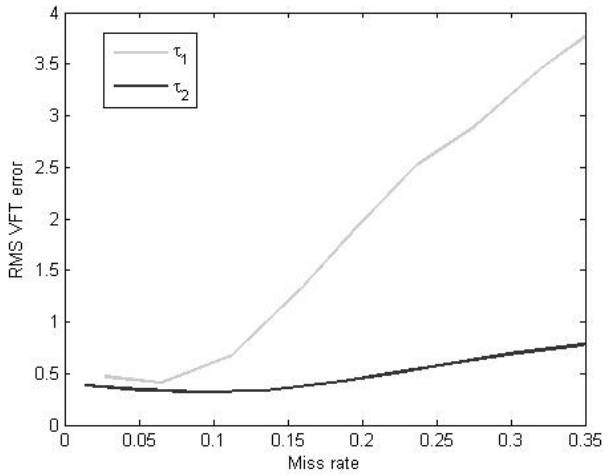


Figure 13: Trade-off between miss-rate and VFT error with a constant budget allocation.

to the fact that the PBS algorithm aggressively compensates with higher budgets in the event of missed deadlines, whereas there are no similar adaptations in the case of a static allocation. Furthermore, computation from missed jobs accumulate, exacerbating VFT errors. Overall, adaptive budget-allocation based on PBS performs better than a tuned static budget allocation, delivering the same miss rate with lower VFT errors and so better budget utilization.

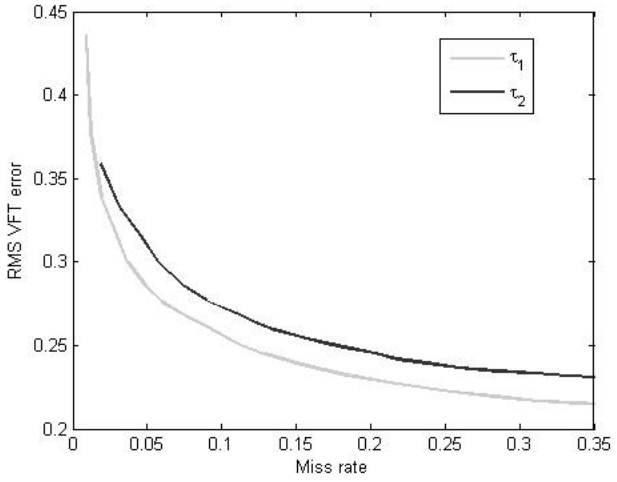


Figure 14: Trade-off between miss-rate and VFT error with PBS-based budget allocation.

CHAPTER IV

TWO-STAGE PREDICTION

In this chapter, the Two-Stage Prediction (TSP) algorithm is presented. The TSP algorithm is a generalization of the PBS algorithm described in Chapter 4 that offers stronger timeliness guarantees with weaker restrictions on job-execution times and can potentially lead to more efficient CPU-budget allocation.

The PBS algorithm described in Chapter 3 improves on a number of previous adaptive budget-allocation algorithms by adapting at a faster rate. Specifically, the PBS algorithm performs asynchronous adaptations at reservation-period boundaries, which allows budget allocations to be adapted on time even when a job deadline is missed. Furthermore, it is proved that asynchronous budget allocation based on the PBS algorithm results in a stable job queue when the execution times of successive jobs are i.i.d.

However, execution times of successive jobs measured from multimedia workloads were found to be correlated and not i.i.d. Experimental results presented in Chapter 3 show that PBS is still stable and performs well, even with workloads where the i.i.d condition is not satisfied. On the other hand, it should be possible to improve on the performance of the moving-average filters used in the PBS algorithm by exploiting the correlation between job-execution times. Also, it is possible in some applications to exploit domain knowledge for more accurate predictions. For example, metadata such as pixel count, byte count, and macroblock count and other indicators can be used to predict frame decoding-times in video-decoding workloads [63][17]. The PBS algorithm and software architecture described in the Chapter 3 do not have the flexibility to allow such prediction techniques to be used.

Furthermore, it is desirable to have a stronger guarantee from a budget-allocation algorithm than the bounded-queue-length guarantee offered by PBS. For example, it is possible for the job-queue length of a task to be bounded and remain constant at two. However, for the queue-length to remain constant at two implies that the deadline of every job is likely missed.

In some of the previous approaches discussed in Section 2.4.2, [72][26], CPU-budget is allocated with the goal of tracking a target miss rate or probability of a deadline miss. In the latter case, these approaches are based on computing sample percentiles of the execution times of recently completed jobs to approximate the inverse cumulative-distribution function (CDF) of the execution times of future jobs. However, it is inherently assumed in these approaches that the sample distribution of the execution times of previous jobs can be used to estimate the ensemble distribution of execution times of future jobs. The execution times of successive jobs are not necessarily independent or identically distributed and therefore, this assumption is not always valid. Even if percentile-based budget-allocation algorithms are stable, such algorithms can benefit from an initial prediction step to reduce the uncertainty in the execution times being predicted. The TSP algorithm is designed to provide a similar guarantee: an upper bound on the probability of a deadline miss. However, this guarantee is provided without the strict i.i.d requirement on job-execution times.

The overall structure of the TSP algorithm is shown in Figure 15. The first stage provides predictions of job-execution times, while the second stage uses the predictions and additional feedback from the system to compute the budget allocation, $Q[k]$. The budget allocation is computed such that an upper bound can be imposed on the probability that the most recently released job will miss the corresponding deadline. As jobs complete, the execution times of the completed jobs, $c[j]$, are fed back into the first stage. These execution times are used by the first stage to update the predictor state, and to update the estimates of the prediction-error statistics.

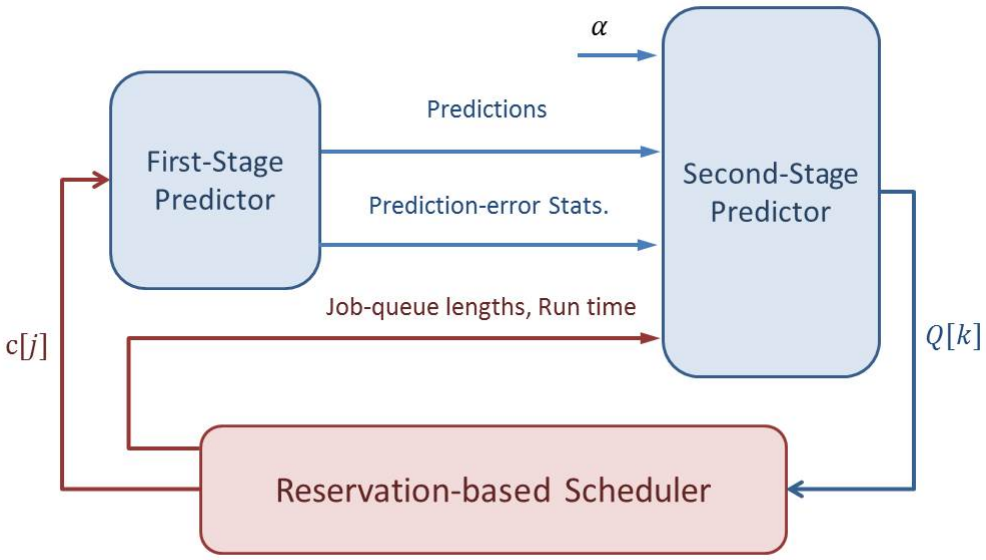


Figure 15: Overall structure of the Two-Stage Prediction algorithm.

In the software architecture presented in this chapter, the first stage is local to each application. This architecture allows the application designer to choose the prediction method and make the desired trade-off between prediction overhead and accuracy. While this approach allows application-specific domain knowledge to be exploited for better prediction accuracy, the scope of the work presented here is limited to a generic prediction algorithm. Specifically, the prediction algorithm presented in this chapter is based on exploiting autocorrelation. For certain workloads, this prediction algorithm offers greater accuracy than a simple moving average. More accurate predictions of job-execution times allow for a more efficient allocation of CPU budget such that the excess CPU capacity allocated to each task can be reduced without an increase in the probability of a deadline miss.

Some initial basic results for the TSP algorithm were first presented in [9]. The remainder of this chapter is organized as follows. First, the task model and CPU-reservation model is presented in Section 4.1. A generic first-stage prediction algorithm, called the LMS-MA Hybrid predictor, is presented in Section 4.2. The second

stage of the TSP algorithm is presented in Section 4.3. Implementation-related details of TSP are presented in Section 4.4. Experimental results are presented in Section 4.5.

4.1 Revisiting the Task Model and CPU-Reservation Model

The task model and scheduling model used for the TSP algorithm is identical to the model considered in Section 3.1. Namely, the system hosts some number of soft-real-time tasks, and each task consists of a series of jobs. These jobs have known periodic release times and deadlines and unknown execution times, $c[j]$. The fixed interval of time between successive release times is referred to as the task period. Each job is assigned an index equal to the index of the task period in which the job is released. The job released in the j^{th} task period is denoted J_j.

Each task is periodically allocated time to run on the CPU over periodic intervals. The allocated time is referred to as CPU budget, and the periodic interval is referred to as a reservation period. Reservation periods are integer submultiples of the task period. If the total budget, $Q_{total}[j]$, allocated over the j^{th} task period is greater than the job-execution time, $c[j]$, and there are no earlier unfinished jobs, the deadline is met. If the allocated budget is less than the job-execution time, the deadline is missed.

The deadline of a job, J_j, is set to the release time of following job, J_{j+1}. Therefore, when a job-deadline is missed, following jobs that are released are queued until all previously-released jobs are completed. In order for a job deadline to be met, sufficient budget must be allocated to accommodate the execution time of the corresponding job and the remaining execution time of all earlier-released jobs that are not yet completed. The total CPU budget needed to complete all queued jobs and the most recently released job, J_j, is referred to as the total computational load, $L[j]$.

Whether or not a job, J_j, will meet the corresponding deadline depends on the

total allocated budget, $Q_{total}[j]$, compared to the total computational load, $L[j]$. These terms are discussed in greater detail in Section 4.3.1.

4.2 First-Stage Prediction

The purpose of the first stage of the TSP algorithm is to provide predictions of future job-execution times. The accuracy of the first-stage predictor affects the overall performance of the TSP algorithm. A low variance in the first-stage prediction error can reduce the budget allocation needed to impose a specific bound on the probability of a deadline miss.

In addition to producing accurate predictions, the prediction algorithm should have low computational requirements. The prediction operation is performed in the execution context of jobs and as a part of jobs. As a result, the first-stage predcitor contributes to job-execution times and consumes a portion of the allocated CPU budget. The overhead of a prediction algorithm should not be so great as to outweigh any reduction in budget allocation enabled by the accuracy of that algorithm.

In the remainder of this section, a generic first-stage prediction algorithm is presented: the *LMS-MA Hybrid* predictor. This algorithm consists of an array of moving averages and normalized variable-step LMS filters. LMS filters are more accurate at predicting job-execution times when the execution times of successive jobs are correlated. Moving averages are more accurate when there is little or no correlation between the execution times of successive jobs. Job-execution times from multimedia workloads can be both correlated or uncorrelated depending on the type, format or even the content of the data being processed. The LMS-MA Hybrid predictor is designed with the goal of performing well in both cases. In addition, the LMS-MA Hybrid predictor does not require any a priori knowledge or parameter tuning making it robust and easy to use. In the discussion presented in the following sections, job-execution times are denoted $c[j]$ and the predicted execution times are denoted

$\hat{c}[j]$. Prediction errors are denoted $e[j]$ and are computed such that:

$$c[j] = \hat{c}[j] + e[j]. \tag{45}$$

4.2.1 Array of Moving Averages

In the case of a moving average, the predicted value, $\hat{c}[j]$, is computed as the sample mean of the execution times of the N most-recently-completed jobs. The length, N, is a tunable parameter. Longer moving averages allow for better noise suppression but are slower to respond to changes in the filtered signal. The optimum length that minimizes the prediction error depends on the the job-execution times of the specific workload. However, it is not desirable to require the user to tune the length of the moving average for each workload.

To eliminate the need for tuning, an array of moving averages can be used. With an array of moving averages, the execution time of the next job is predicted with multiple filters, each of a different length. To reduce the overhead of computing the output of longer filters, the output of shorter filters are used as a part of the computation. Let B denote the number of filters in the array, let $\hat{c}_b[j]$ denote the output of the b^{th} moving-average, and let N_b denote the corresponding length. The filter outputs are defined as follows:

$$\hat{c}_b[j] = \begin{cases} 0 & \text{for } b = 0 \\ \dfrac{1}{N_b} \displaystyle\sum_{i=1}^{N_b} c[j-i] & \text{for } b = 1, 2, ..., B \end{cases}. \tag{46}$$

When a job is completed and the corresponding execution time is known, the prediction error, $e_b[j]$, is computed for each moving average. Then, the approximate mean and variance of the prediction error is computed for each filter using an exponentially-weighted moving average (EWMA):

$$e_b[j] = c[j] - \hat{c}_b[j], \tag{47}$$

$$\hat{\mathbb{E}}\left(e_b[j]\right) = \gamma \hat{\mathbb{E}}\left(e_b[j-1]\right) + (1-\gamma) e_b[j], \tag{48}$$

$$\hat{\mathrm{Var}}\left(e_b[j]\right) = \gamma \hat{\mathrm{Var}}\left(e_b[j-1]\right) + (1-\gamma)\left(e_b[j] - \hat{\mathbb{E}}\left(e_b[j]\right)\right)^2. \tag{49}$$

An EWMA is used to compute the error mean and error variance because it was found to perform better overall in the TSP algorithm than a uniformly-weighted moving average. Also, the computational and memory overhead of an EWMA is lower than that of a uniformly-weighted moving average.

Once the error mean and variance is computed for all the filters, the next output of the predictor is set to the output of the filter with the least error variance. Let $b*$ denote the index of the filter with the minimum error variance. The error mean and variance of the predictor is set to the error mean and variance of the chosen filter:

$$\hat{c}[j] = \hat{c}_{b*}[j], \tag{50}$$

$$\hat{\mathbb{E}}\left(e[j]\right) = \hat{\mathbb{E}}\left(e_{b*}[j]\right), \tag{51}$$

$$\hat{\mathrm{Std}}\left(e[j]\right) = \sqrt{\hat{\mathrm{Var}}\left(e_{b*}[j]\right)}. \tag{52}$$

4.2.2 The Normalized Variable-Step LMS Algorithm

In certain workloads, there is a strong correlation between the execution-times of successive jobs. This correlation is not fully exploited by a moving average. In contrast, the Wiener filter is specifically designed to exploit autocorrelation.

A Wiener filter is the optimum finite impulse response (FIR) filter "that provides the minimum mean-square-error estimate of a given process by filtering a set of observations of a statistically related process"[39]. For the purposes of execution-time prediction, a Wiener filter is a linear predictor that provides the least mean-square error (MSE) estimate of the execution time of a future job by filtering the execution times of previous jobs. Let w_i denote the coefficients of the Wiener filter. The predicted value is computed as follows:

$$\hat{c}[j] = \sum_{i=1}^{N} w_i c[j-i]. \tag{53}$$

The wiener filter minimizes the MSE, $\mathbb{E}\left((c[j] - \hat{c}[j])^2\right)$. However, computing the coefficients of the Wiener filter, $\vec{w}$, requires knowledge of the autocorrelation matrix of job-execution times. It is not practical to assume that the autocorrelation matrix is known or that the matrix can be dynamically computed for each workload.

An alternative to directly applying the Wiener filter is to use an adaptive filter based on the LMS algorithm [39]. The LMS algorithm is a steepest-decent algorithm for searching for an optimum set of coefficients with the least MSE. The MSE is a quadratic function of the filter coefficients. The search for the optimum set of coefficients involves iteratively correcting the coefficients of an adaptive filter in the approximate direction of largest decrease in the MSE. The direction and magnitude of steepest decent is not precisely known, and the corrections are approximate. However, on average, the coefficients of the adaptive filter converge towards the coefficients of the the Wiener filter [39].

The predicted value and the prediction error are computed in the same way as for the Wiener filter:

$$\hat{c}[j] = \sum_{i=1}^{N} w_i[j]c[j-i] \qquad\qquad e[j] = c[j] - \hat{c}[j]. \tag{54}$$

However, as new values of the prediction error become known, the filter coefficients are adjusted as follows:

$$w_i[j+1] = w_i[j] + \beta \left(\frac{-c[j-i]e[j]}{\sum_{k=1}^{N} c^2[j-k]} \right), \tag{55}$$

where β is the step size and N is the length of the filter. This equation for adjusting the filter coefficients is based on the *Normalized LMS algorithm*, and the sum of squares in the denominator is the normalizing factor. The normalized LMS algorithm converges in the mean square if $0 < \beta < 2$ [39].

The step size, β, affects the rate of convergence of the filter coefficients as well as

the steady-state MSE. When the filter coefficients converge, the coefficients fluctuate about the optimum because the corrections to the coefficients are approximate. As a result, the steady-state MSE is larger than the ideal MSE by an amount referred to as the *excess mean-square error* [39]. Both the convergence rate and the excess mean-square error increase with the step size.

Since it is not desirable to require that the step size be tuned, the Variable Step (VS) adaptive filter algorithm is used[38]. In this algorithm, there is a separate step size, $\beta_i[j]$, for each filter coefficient and these step sizes are adapted dynamically. When a filter coefficient is further from the optimal value, the corresponding step size is increased so that the coefficient converges faster. When the filter coefficient is closer to the optimal value, the step size is decreased so that any misadjustment in the coefficient is smaller and the resulting excess mean-square error is smaller.

As mentioned earlier, the MSE is a quadratic function of the filter coefficients. At the minimum, the gradient of the MSE is zero. Whether or not a coefficient is close to the optimal value can be determined by the frequency of zero crossings in the corresponding component of the MSE gradient. Equivalently, "closeness" of a coefficient to the optimum value can be determined by the frequency of sign changes in the corresponding component of the MSE gradient. The component of the MSE gradient corresponding to the coefficient w_i is approximated as follows:

$$\nabla_i \left(\mathbb{E} \left(e^2[j] \right) \right) = \frac{\partial \mathbb{E} \left(e^2[j] \right)}{\partial w_i[j]} \approx -2e[j]c[j-i]. \tag{56}$$

If the sign of a component of the MSE gradient changes repeatedly on S_0 successive steps, it is assumed that the corresponding filter coefficient is close to the optimal value and the step size is halved. If the sign stays constant on S_1 successive steps, it is assumed that the filter coefficient is still far from the optimal value and the step size is doubled. Based on the discussion in [38], S_1 is set to 3 and S_0 is set to $(S_1 - 1) = 2$. For reasons of stability, the step sizes are bounded above by $\beta_{max} = 1.5$. A lower bound, $\beta_{min} = 0.0125$, is also applied. It is assumed that the filter coefficients are

initially far from the optimal value and therefore, the step sizes are initialized to the maximum value, β_{max}. The normalized VS LMS algorithm is summarized in Figure 16.

$$\hat{c}[j] = \sum_{i=1}^{N} w_i[j]c[j-i] \qquad\qquad e[j] = c[j] - \hat{c}[j].$$

$$w_i[j+1] = w_i[j] + \beta_i[j] \left(\frac{-c[j-i]e[j]}{\sum_{k=1}^{N} c^2[j-k]} \right),$$

$$\beta_i[j+1] = \begin{cases} \frac{1}{2}\beta_i[j] & \text{iff } \beta_i[j] > \beta_{min} \text{ and } \operatorname{sgn}\left(\nabla_i \left(\mathbb{E}\left(e^2[j]\right)\right)\right) \text{ changes } S_0 \text{ times} \\ 2\beta_i[j] & \text{iff } \beta_i[j] < \beta_{max} \text{ and } \operatorname{sgn}\left(\nabla_i \left(\mathbb{E}\left(e^2[j]\right)\right)\right) \text{ stays constant } S_1 \text{ times} \\ \beta_i[j] & otherwise \end{cases}$$

$$\nabla_i \left(\mathbb{E}\left(e^2[j]\right)\right) \approx -2e[j]c[j-i].$$

Figure 16: Summary of the normalized VS LMS algorithm.

4.2.3 Array of Adaptive Filters

The accuracy of the normalized VS-LMS algorithm is sensitive to the filter length as well. The filter-length used by the predictor may be less than the length of the FIR system being modeled even though it is assumed in the design of the filter that the two lengths are the same. Furthermore, the optimum filter length may change with time [62]. However, in the same way as for the step size, it is not desirable to require that the filter length be tuned for each workload. Instead, the filter length should be adapted.

Variable-Length LMS algorithms have been explored in previous research. In one approach presented in [62], an M-tap adaptive filter is represented as a cascade of K P-tap adaptive filters where $M = KP$. At each time step, the MSE is estimated at the output of each stage of the cascade. The output of the overall system is set to the

output of the stage with the least MSE. A similar cascading approach is presented in [55] to address the problem of slow convergence in long filters. In another approach presented in [20], the system simultaneously maintains three adaptive filter, $\hat{h}_1$, $\hat{h}_2$, and $\hat{h}_3$ of lengths N_1, $N_2 = (N_1 + 1)$, and $N_3 = (N_1 + 2)$, respectively. At each time step, the MSE is dynamically estimated for all three filters to determine if the filter length, N_2, is a minimum for the MSE. If N_2 is not a minimum, the length of all three filters are adapted in the direction of greatest decrease in the MSE.

A cascade of LMS filters as proposed in [62] does not have the same flexibility as a single large LMS filter and can have limited performance. On the other hand, changing the filter length dynamically as proposed in [55] can result in transients. Transients can negatively affect performance and limit how often the filter length can be adapted.

As an alternative to these two approaches, an array of LMS filters is proposed. This filter array is similar to the array of moving-average filters discussed in Section 4.2.1. The array consists of multiple Normalized VS-LMS filters similar to the ones described in Section 4.2.2. Each filter in the array has a different length but the length of the individual filters are kept constant. The output of the array is set to the output of the filter with the least MSE.

Computationally, this filter-array approach can be significantly more expensive than the cascade approach or gradient-descent approach. Another disadvantage is that the lengths of the filters in the array are not flexible. However, with this approach, there are no transients due to changes in the filter length. Switching is done at the output of the adaptive filters and not in the feedback path, and therefore, can be performed independent of the adaptations in the filter coefficients. Also, the additional computational overhead associated with this approach may be acceptable if the decrease in budget allocation enabled by the accuracy of this approach can outweigh the larger overhead.

4.2.4 The LMS-MA Hybrid Predictor

When the execution times of successive jobs are correlated, the LMS filter can offer better prediction accuracy and lower variance in the prediction error. When there is little or no correlation, a moving average can perform better. In both the array of moving averages described in Section 4.2.1 and in the array of LMS filters described in Section 4.2.3, the output of the array is set to the output of the filter with the least estimated MSE. Similarly, these two filter arrays can be merged into a larger array, and the output of this larger array can be set to the output of the filter with the least estimated MSE. The goal of using this combined array is to have the advantage of both types of filters and to handle both types of workloads with correlated or uncorrelated job-execution times. This combined array is called the *LMS-MA Hybrid predictor*. The performance of this algorithm in terms of overhead and prediction accuracy is discussed in Section 4.5.4.

4.3 The Second Stage

The purpose of the second stage of the TSP algorithm is to use the output of the first-stage predictor and compute a valid budget allocation such that an upper bound can be imposed on the probability of a deadline miss.

In this section, the second stage of the TSP algorithm is developed. An equation is derived for budget allocation, and Cantelli's inequality is used to prove the upper bound on the probability of a deadline miss. Then, this equation for budget allocation is modified to exploit the output of the first stage predictor. As mentioned earlier, the purpose of the first-stage predictor is to reduce the uncertainty in job-execution times in order to reduce the average budget allocation necessary to impose a specific upper bound. The equation for budget allocation is further modified to produce valid budget allocations in cases of severe underpredictions. Then, multi-step prediction is discussed, and the equation for budget allocation is further modified to make it

practically feasible in cases of missed deadlines and job queueing. The final form of the budget-allocation equation is used as the second stage of the TSP algorithm. Before proceeding with the development of the second stage, expressions are derived for total allocated budget and total computational load.

4.3.1 Allocated Budget & Computational Load

As mentioned in Section 4.1, the total computational load, $L[j]$, is the total CPU budget needed to complete all queued jobs and the most recently released job. Whether or not a job, J_j, will meet the corresponding deadline depends on the total allocated budget, $Q_{total}[j]$, compared to total computational load, $L[j]$.

Before presenting expressions for $L[j]$ and $Q_{total}[j]$, some of the notation from Section 3.3 is repeated here for convenience. Let K denote the number of reservation periods in a task period. The index of the reservation period, k, and the task period, j, are related such that $j = \lfloor k/K \rfloor$. The indices k_{first} and k_{last} denote the first and last reservation periods of the task period:

$$k_{first} = jK \qquad\qquad k_{last} = jK + K - 1. \tag{57}$$

The variable $l[k]$ denotes the number of jobs queued at the start of the k^{th} reservation period. The variable $c_{min}[k]$ denotes the CPU budget already consumed by the currently running job before the start of the k^{th} reservation period. The variable $Q[k]$ denotes the CPU budget allocated in the k^{th} reservation period. Finally, the following notation is used to denote the summation of job-execution times:

$$\overset{j_2}{\underset{j_1}{\mathrm{C}}} = \begin{cases} \displaystyle\sum_{j=j_1}^{j_2} c[j] & \text{iff } j_2 \geq j_1 \\ 0 & \text{otherwise} \end{cases}. \tag{19 revisited}$$

As described in Section 3.3.2, the computational load, $L[j]$, at the start of the j^{th} task period consists of the execution times of all queued jobs minus any CPU-time

already consumed:

$$L[j] = \overset{j}{\underset{j-l[k_{first}]+1}{\mathrm{C}}} - c_{min}[k_{first}]. \tag{58}$$

Equation (58) is similar to (21) except that $L[j]$ corresponds to the computational load from all queued jobs and not just the first l jobs.

Also, in Section 3.3.2, the total budget allocated in the j^{th} task period, $Q_{total}[j]$, is expressed as the sum of the budget allocated in the first $(K-1)$ reservation periods of the task period and the budget allocated in the final reservation period, $Q[k_{last}]$:

$$Q_{total}[j] = \sum_{k=k_{first}}^{k_{last}-1} Q[k] + Q[k_{last}]. \tag{22 revisited}$$

Expanding the summation based on (23), $Q_{total}[j]$ is expressed as follows:

$$Q_{total}[j] = \overset{j-l[k_{last}]}{\underset{j-l[k_{first}]+1}{\mathrm{C}}} + c_{min}[k_{last}] - c_{min}[k_{first}] + Q[k_{last}]. \tag{59}$$

4.3.2 Bounding the Probability of a Missed Deadline

Theorem 2. *Let* $\mathrm{Std}(X)$ *denote the standard deviation of a random variable* X*, let* $\rho_{miss,j}$ *denote the probability of the event that the deadline of job* J_j *is missed, and let* α *denote a user-defined parameter. If the budget allocation in reservation period* k *is computed as follows:*

$$Q[k] = \frac{\mathbb{E}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{C}}}\right) + \alpha\ \mathrm{Std}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{C}}}\right) - c_{min}[k]}{K - (k \mod K)}, \tag{60}$$

then the probability, $\rho_{miss,j}$*, is bounded above by* $\dfrac{1}{1+\alpha^2}$.

The α parameter in Equation 60 is similar to the α parameter in the PBS algorithm. As α increases, the budget allocation increases and the bound on the probability of a deadline miss decreases. Therefore, the α parameter can be used to trade off budget utilization for timliness.

Proof. The probability that job J_j misses the corresponding deadline, $\rho_{miss,j}$, is equal to the probability that the total budget allocated in the j^{th} task period, $Q_{total}[j]$, is less than the total computational load, $L[j]$.

$$\rho_{miss,j} = \Pr\left(L[j] > Q_{total}[j]\right). \tag{61}$$

Substituting for $L[j]$ and $Q_{total}[j]$ with the corresponding expressions, $\rho_{miss,j}$ can be expressed as follows:

$$\rho_{miss,j} = \Pr\left(\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right) > \left(c_{min}[k_{last}] + Q[k_{last}]\right)\right). \tag{62}$$

Evaluating (60) at $k = k_{last}$, the denominator becomes one:

$$Q[k_{last}] = \mathbb{E}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right) + \alpha\ \mathrm{Std}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right) - c_{min}[k_{last}]. \tag{63}$$

Substituting for $Q[k_{last}]$ in (62) and simplifying, $\rho_{miss,j}$ can be expressed as follows:

$$\begin{aligned}\rho_{miss,j} =& \Pr\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}} > \left(\mathbb{E}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right) + \alpha\ \mathrm{Std}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right)\right)\right) \\ =& \Pr\left(\frac{\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}} - \mathbb{E}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right)}{\mathrm{Std}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right)} > \alpha\right).\end{aligned} \tag{64}$$

Since job-execution times are strictly positive, the summation, $\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}$, is also strictly positive and the corresponding probability distribution is one sided. Applying Cantelli's inequality [56], the probability of a job missing the corresponding deadline is bounded above by $\dfrac{1}{1+\alpha^2}$:

$$\rho_{miss,j} = \Pr\left(\frac{\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}} - \mathbb{E}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right)}{\mathrm{Std}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{C}}}\right)} > \alpha\right) \leq \frac{1}{1+\alpha^2}. \tag{65}$$

□

Let $\rho_{desired}$ denote the desired value of the probability of a job missing the deadline. Then, the value of α that will bound the probability accordingly can be computed as follows:

$$\alpha = \sqrt{\left(\frac{1}{\rho_{desired}} - 1\right)}. \tag{66}$$

4.3.3 Using the Output of the First-Stage Predictor

Cantelli's inequality, a one-sided variant of the Chebyshev inequality, gives a conservative upper bound on the probability of a deadline miss (65). Therefore, directly applying Cantelli's inequality as done in (60) may result in higher budget allocations than necessary. If the variance in the first-stage prediction error is lower than the variance in job-execution times, the output of the first-stage predictor can be used to reduce the budget allocation.

As mentioned in Section 4.2, $\hat{c}[j]$ denotes a prediction of the execution time, $c[j]$, and $e[j]$ denotes the corresponding prediction error such that

$$c[j] = \hat{c}[j] + e[j]. \tag{45 revisited}$$

Similar to the notation for summation of job-execution times, let the summation of $\hat{c}[j]$ and $e[j]$ over a set of jobs be denoted as follows:

$$\overset{j2}{\underset{j1}{\hat{\mathrm{C}}}} \triangleq \begin{cases} \displaystyle\sum_{j=j1}^{j2} \hat{c}[j] & \text{for } j2 \geq j1 \\ 0 & \text{otherwise} \end{cases} \qquad \overset{j2}{\underset{j1}{\mathrm{e}}} \triangleq \begin{cases} \displaystyle\sum_{j=j1}^{j2} e[j] & \text{for } j2 \geq j1 \\ 0 & \text{otherwise} \end{cases}. \tag{67}$$

Substituting for the execution time with the predicted value and prediction error, Equation 60 for budget allocation can be restated as follows:

$$Q[k] = \frac{\overset{j}{\underset{j-l[k]+1}{\hat{\mathrm{C}}}} + \mathbb{E}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{e}}}\right) + \alpha \ \mathrm{Std}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{e}}}\right) - c_{min}[k]}{K - (k \mod K)}. \tag{68}$$

The predicted value is known, whereas the prediction error is unknown. Because $\hat{c}[j]$ is known, the corresponding mean is $\hat{c}[j]$ itself and the standard deviation of $\hat{c}[j]$ is zero.

Corollary 1 (Corollary to Theorem 2)**.** *If the budget allocation in reservation period k is computed according to Equation 68 and the predicted value, $\hat{c}[j]$, is bounded above, then the probability, $\rho_{miss,j}$, is bounded above by* $\dfrac{1}{1+\alpha^2}$.

Proof. With similar substitutions of $c[j]$ with $\hat{c}[j]$ and $e[j]$ in (62), the probability of job J_j missing the deadline can be expressed as follows:

$$\rho_{miss,j} = \Pr\left(\left(\overset{j}{\underset{j-l[k_{last}]+1}{\hat{\mathrm{C}}}} + \overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right) > (c_{min}[k_{last}] + Q[k_{last}])\right). \tag{69}$$

Evaluating (68) at $k = k_{last}$,

$$Q[k_{last}] = \overset{j}{\underset{j-l[k_{last}]+1}{\hat{\mathrm{C}}}} + \mathbb{E}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right) + \alpha\ \mathrm{Std}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right) - c_{min}[k_{last}].$$

Substituting for $Q[k_{last}]$ in (69), the probability expression is reduced as follows:

$$\rho_{miss,j} = \Pr\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}} > \left(\mathbb{E}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right) + \alpha\ \mathrm{Std}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right)\right)\right) \tag{70}$$

$$= \Pr\left(\frac{\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}} - \mathbb{E}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right)}{\mathrm{Std}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right)} > \alpha\right). \tag{71}$$

Since the predicted values are bounded above and execution times are strictly positive, the prediction errors, $e[j]$, have a one-sided probability distribution. Therefore, applying Cantelli's inequality, the probability of J_j missing the deadline can still be upper-bounded at $1/(1+\alpha)^2$ in the same way as in (65):

$$\rho_{miss,j} = \Pr\left(\frac{\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}} - \mathbb{E}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right)}{\mathrm{Std}\left(\overset{j}{\underset{j-l[k_{last}]+1}{\mathrm{e}}}\right)} > \alpha\right) \leq \frac{1}{1+\alpha^2}. \tag{72}$$

□

Equation 68 is a generalization of (60) where the predicted execution time, $\hat{c}[j]$, is identically set to zero and $e[j] = c[j]$. However, with an effective first-stage prediction algorithm, the variance in the prediction error may be significantly smaller than the

variance in the execution time. A smaller variance in the prediction error allows for a smaller budget allocation while providing the same upper bound on the probability of missing the deadline.

Corollary 1 requires an upper bound on the predicted values of job-execution times. This condition must be satisfied by the first-stage-prediction algorithm. In the case of the moving-average filter or LMS filter discussed in Section 4.2, predicted values are computed as weighted averages of previous job-execution times. For both moving averages and for stable LMS filters, the filter coefficients are bounded. Since job-execution times are bounded by WCETs, the predicted values are also bounded, and the requirements of Corollary 1 are satisfied.

4.3.4 Handling Severe Underpredictions

In most cases, Equation 68 will produce appropriate values for budget allocation. However, if the predicted value of job-execution time, $\hat{c}[j]$, is significantly smaller than the actual execution time and the value of $c_{min}[k]$ is large, then Equation 68 may produce zero or negative values for budget allocation while there are jobs pending.

This special case can be addressed in the same way as addressed in the PBS algorithm, by using a conditional mean in the estimate of the total computational load. Specifically, the contribution of the currently running job to the computational load should not be approximated with just the mean execution time. Instead, the conditional mean should be used given that this execution time is known to be larger than $c_{min}[k]$. In addition, no budget should be allocated when there are no jobs queued.

Let j_{run} denote the index of the currently running job at the start of the k^{th} reservation period. Based on the discussion in Section 3.3.2, j_{run} is related to k as follows:

$$j_{run} = j - l[k] + 1, \tag{73}$$

where $l[k]$ denotes the length of the job queue. Accordingly, $c[j_{run}]$ is the execution time of the job running at the start of the k^{th} reservation period. Let $\tilde{c}_{cond}[k]$ denote an approximation of the conditional mean of $c[j_{run}]$ given that $c[j_{run}]$ is greater than $c_{min}[k]$:

$$\tilde{c}_{cond}[k] \approx \mathbb{E}\left(c[j_{run}] \mid c[j_{run}] > c_{min}[k]\right). \tag{74}$$

Equation (68) for budget allocation is modified as follows:

$$Q[k] = \begin{cases} \dfrac{\tilde{c}_{cond}[k] + \hat{\mathrm{C}}_{j-l[k]+2}^{j} + \mathbb{E}\left(\hat{\mathrm{e}}_{j-l[k]+2}^{j}\right) + \alpha \ \mathrm{Std}\left(\hat{\mathrm{e}}_{j-l[k]+1}^{j}\right) - c_{min}[k]}{K - (k \mod K)} & l[k] > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{75}$$

By definition, $\tilde{c}_{cond}[k]$ is greater than $c_{min}[k]$, and $(\tilde{c}_{cond}[k] - c_{min}[k])$ is always positive. Therefore, the budget allocation, $Q[k]$, as computed by Equation 75, is strictly positive for job-queue lengths greater than zero, addressing the underprediction problem.

Furthermore, the previous upper bound on the probability of a deadline miss is still valid. By definition, $\tilde{c}_{cond}[k]$ is greater than or equal to $\mathbb{E}(c[j_{run}])$. For the case when the queue length is greater than zero, budget allocation defined in (75) is greater than the budget allocation defined in (68). With larger budget allocations, the probability of J_j missing the deadline is lower. For the case when the queue length is equal to zero, the most recently released job, J_j, has already completed before the corresponding deadline and the probability of missing the deadline is zero. For both cases, the probability of a deadline miss is still bounded above by $1/(1+\alpha^2)$.

4.3.5 Approximating the Conditional Mean

To compute an estimate of the conditional mean defined by (74), the probability distribution is needed. As in the case of the PBS algorithm, the distribution of job-execution times is approximated with a translated exponential distribution. The

translated exponential distribution is an appropriate approximation for long jobs because job-execution times of most workloads have probability distributions with long tails. Also, an advantage of using an exponential distribution is that the conditional mean can be computed with little overhead due to the memorylessness property.

A translated exponential distribution is defined by two parameters: the translation amount, denoted c_{trans}, and the rate parameter of the exponential distribution, λ. These parameters can be assigned values such that the mean and standard deviation of the resulting distribution are equal to the mean and standard deviation of job-execution times. Both the mean and standard deviation of an exponentially distributed random variable are equal to λ^{-1}. Since the standard deviation is unaffected by translation, $\lambda^{-1}[j]$ is assigned a value equal to the standard deviation, $\mathrm{Std}\left(c[j]\right)$. Accordingly, the translation amount is computed as follows:

$$c_{trans}[j] = \mathbb{E}\left(c[j]\right) - \mathrm{Std}\left(c[j]\right). \tag{76}$$

Because the exponential distribution is memoryless, $\lambda[j]$ does not change with conditioning. The only affect of conditioning on the mean is a change in the translation amount. The conditional mean of job-execution times is approximated as follows:

$$\mathbb{E}\left(c[j] \mid c[j] > c_{min}\right) \approx \begin{cases} c_{trans}[j] + \mathrm{Std}\left(c[j]\right) & \text{iff } c_{trans}[j] \geq c_{min} \\ c_{min} + \mathrm{Std}\left(c[j]\right) & \text{otherwise} \end{cases} \tag{77}$$

Substituting $c[j]$ and $c_{trans}[j]$ with the corresponding expansions, the above expressions are reduced as follows:

$$c_{trans}[j] = \hat{c}[j] + \mathbb{E}\left(e[j]\right) - \mathrm{Std}\left(e[j]\right). \tag{78}$$

$$\mathbb{E}\left(c[j] \mid c[j] > c_{min}\right) \approx \begin{cases} \hat{c}[j] + \mathbb{E}\left(e[j]\right) & \text{iff } c_{trans}[j] \geq c_{min} \\ c_{min} + \mathrm{Std}\left(e[j]\right) & \text{otherwise.} \end{cases} \tag{79}$$

Finally, based on (74), $\tilde{c}_{cond}[k]$ is expressed as follows:

$$\tilde{c}_{cond}[k] = \begin{cases} \hat{c}[j_{run}] + \mathbb{E}\left(e[j_{run}]\right) & \text{iff } c_{trans}[j_{run}] \geq c_{min}[k] \\ c_{min}[k] + \text{Std}\left(e[j_{run}]\right) & \text{otherwise.} \end{cases} \tag{80}$$

4.3.6 Multi-step Prediction

When a job deadline is missed, one or more jobs are queued and the queue-length is more than one. In this case, evaluating Equation 75 for budget allocation requires multi-step prediction; the execution times of more recently-released jobs must be predicted while the execution time of the currently-running job is still unknown. Furthermore, (75) requires an estimate of the expected value and standard deviation of the sum of the errors across these prediction steps.

For the LMS-based prediction algorithm described in Section 4.2.2, job-execution times are modeled as an Auto-Regressive Moving-Average (ARMA) process. Given previous values of an ARMA process, the LMS filter can be used to predict the next value of that process. When predicting the value of the process more than a single step after the most recent observation, a separate filter is required for each additional step. Also, the further out the prediction (the larger the number of prediction steps), the lower the prediction accuracy. Such an approach is not feasible. On the other hand, for prediction algorithms based on moving averages, job-execution times are modeled as a mean-ergodic and variance-ergodic process. Regardless of the number of prediction steps, the predicted value and error variance for the moving average remain the same.

A common design principle in computing systems is to optimize for the common case. In most cases, jobs meet corresponding deadlines, the queue length is one, and only a single prediction step is necessary. Therefore, to predict the execution time of the currently-running job, the full LMS-MA hybrid algorithm can be used. On the other hand, for predictions of execution times of jobs released later, the array

of moving averages can be used. Since queue lengths greater than one should occur less frequently, the accuracy of multi-step prediction should have a limited impact on performance overall.

Let $\hat{c}_0[j]$ denote the single-step prediction of $c[j]$ and let $e_0[j]$ denote the corresponding prediction error. Similarly, let $\hat{c}_l[j]$ denote the multi-step prediction of $c[j]$ regardless of the number of prediction steps and let $e_l[j]$ denote the corresponding prediction error. In the implementation of the LMS-MA Hybrid algorithm, $\hat{c}_0[j]$ can be set to the output of the least-MSE filter of the complete array, and $\hat{c}_l[j]$ can be set to the output of the least-MSE moving average.

The second issue is concerning the expected value and standard deviation of the sum of errors across the steps of the multi-step prediction. Prediction errors are random variables. Computing the expected value of a sum of random variables is straightforward. However, the standard deviation is the square-root of the variance, and the variance of a sum of random variables, $X_1,\ X_2,\ ..., X_N$ is the sum of the covariance of each pair of variables in that sum [51]:

$$\mathrm{Var}\left(\sum_{i=1}^{N} X_i\right) = \sum_{i=1}^{N}\sum_{j=1}^{N} \mathrm{Cov}(X_i, X_j). \tag{81}$$

Therefore, to compute the variance of the sum of errors, the autocovariance of the prediction-error process is required.

However, as mentioned above, it should be sufficient in most cases to compute the error variance for a single prediction step. When it is necessary to address multiple prediction steps, it can be assumed that successive values of the prediction-error process are uncorrelated or weakly correlated. Then, the variance of the sum can be approximated with a sum of variance. Based on the results presented in Section 4.5.4, the weak-autocorrelation assumption is not always valid for the prediction-error process. However, since queue lengths greater than one should occur less frequently, this approximation should have a limited impact even when the weak-correlation

assumption does not hold.

Based on the above discussion, the expected value of the execution time of jobs queued after the currently running job is approximated with a moving average in the same way as in the PBS algorithm. The approximate expected value of the sum of the execution times of queued jobs is computed as follows:

$$\hat{\mathbb{E}}\left(\overset{j}{\underset{j-l[k]+2}{\mathrm{C}}}\right) = (l[k]-1)\left(\hat{c}_l[j_{run}] + \hat{\mathbb{E}}(e_l[j_{run}])\right). \tag{82}$$

The approximate standard deviation of the sum of the execution times of all queued jobs is computed as follows:

$$\hat{\mathrm{Std}}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{C}}}\right) = \sqrt{\hat{\mathrm{Var}}(e_0[j_{run}]) + (l[k]-1)\,\hat{\mathrm{Var}}(e_l[j_{run}])}. \tag{83}$$

Finally, given the expressions for $\hat{\mathbb{E}}\left(\overset{j}{\underset{j-l[k]+2}{\mathrm{C}}}\right)$ and $\hat{\mathrm{Std}}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{C}}}\right)$, budget allocation is computed as follows:

$$Q[k] = \begin{cases} \dfrac{\tilde{c}_{cond}[k] - c_{min}[k] + \hat{\mathbb{E}}\left(\overset{j}{\underset{j-l[k]+2}{\mathrm{C}}}\right) + \alpha\hat{\mathrm{Std}}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{C}}}\right)}{K - (k \mod K)} & l[k] > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{84}$$

Equation 84 represents the second stage of the TSP algorithm. A summary of some of the terms used in the TSP algorithm are presented in Figure 17 and a summary of the second stage of the TSP algorithm is presented in Figure 18.

k : index of the current reservation period (RP).
K : # of reservation periods in a task period.
j : index of the current task period, $j = \lfloor k/K \rfloor$.
$l[k]$: # of jobs queued and running in the k^{th} RP.
j_{run} : index of the currently running job, $j_{run} = j - l[k] + 1$
J_j : job released in the j^{th} task period.
$c[j]$: execution time of J_j.
$\hat{c}_0[j]$: single-step prediction of $c[j]$.
$e_0[j]$: prediction error in $\hat{c}_0[j]$, such that $c[j] = \hat{c}_0[j] + e_0[j]$.
$\hat{c}_l[j]$: multi-step prediction of $c[j]$.
$e_l[j]$: prediction error in $\hat{c}_l[j]$, such that $c[j] = \hat{c}_l[j] + e_l[j]$.
$c_{min}[k]$: CPU budget consumed by $J_{j_{run}}$ by the start of the k^{th} RP.
$c_{cond}[k]$: approximate conditional mean of $c[j_{run}]$ given that $c[j_{run}] > c_{min}[k]$
$\hat{\mathbb{E}}(X)$: approximate expected value of a random variable X
$\hat{\mathrm{Var}}(X)$: approximate variance of a random variable X

Figure 17: Summary of terms used for the TSP algorithm.

4.4 Implementing the TSP Algorithm

The implementation of TSP builds on the implementation of PBS described in Section 3.5. At each reservation-period boundary, the budget allocation for each SRT task is computed by a daemon process called the Allocator. Execution-time measurements, budget enforcement, and other privileged operations are performed in a kernel module similar to the PBS module described in Section 3.5.2. SRT tasks and the Allocator interact with the kernel module through special files in the `/proc` file system. Also, the Allocator and the kernel module share large amounts of data efficiently using memory mapping as described in Section 3.5.3.

There are two main differences between the implementations of TSP and PBS. The first is the budget-allocation algorithm as described in the preceding sections. The second is the execution context in which the prediction operation is performed.

$$c_{trans}[j] = \hat{c}_0[j] + \mathbb{E}\left(e_0[j]\right) - \mathrm{Std}\left(e_0[j]\right)$$

$$\tilde{c}_{cond}[k] = \begin{cases} \hat{c}_0[j_{run}] + \mathbb{E}\left(e_0[j_{run}]\right) & \text{iff } c_{trans}[j_{run}] \geq c_{min}[k] \\ c_{min}[k] + \mathrm{Std}\left(e_0[j_{run}]\right) & \text{otherwise.} \end{cases}$$

$$\hat{\mathbb{E}}\left(\overset{j}{\underset{j-l[k]+2}{\mathrm{C}}}\right) = \left(l[k]-1\right)\left(\hat{c}_l[j_{run}] + \hat{\mathbb{E}}(e_l[j_{run}])\right)$$

$$\hat{\mathrm{Std}}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{C}}}\right) = \sqrt{\hat{\mathrm{Var}}(e_0[j_{run}]) + \left(l[k]-1\right)\hat{\mathrm{Var}}(e_l[j_{run}])}$$

$$Q[k] = \begin{cases} \dfrac{\tilde{c}_{cond}[k] - c_{min}[k] + \hat{\mathbb{E}}\left(\overset{j}{\underset{j-l[k]+2}{\mathrm{C}}}\right) + \alpha\hat{\mathrm{Std}}\left(\overset{j}{\underset{j-l[k]+1}{\mathrm{C}}}\right)}{K - (k \mod K)} & l[k] > 0 \\ 0 & \text{otherwise} \end{cases}$$

Figure 18: Summary of the second stage of the TSP algorithm.

In the implementation of PBS, job-execution times are predicted with simpe moving averages in the execution context of the Allocator. In the implementation of TSP, the prediction operation for each SRT task is performed in the execution context of the task itself.

Among the advantages of performing the prediction operation in the SRT task is that different algorithms can be applied to different applications and workloads, and the application designer has flexibility in the choice of the algorithm. The prediction overhead contributes to the execution time of SRT jobs, and the choice of the prediction algorithm allows the application designer to make the appropriate trade-off between accuracy and overhead.

4.4.1 Measurement and Prediction of Job-Execution Times

Among the consequences of performing the prediction operation in the execution context of SRT tasks is a change in the way that job-execution times are measured for the purposes of prediction.

In the implementation of PBS, a job only consists of workload-specific computation. When a job completes and the following job is still not released, the corresponding task is put to sleep until the following task-period boundary. The job-execution time is measured and recorded by the kernel module and stored in a memory location that is accessible to the Allocator. When the Allocator is activated, the execution times of future jobs are predicted based on a running average of the execution times of previous jobs. Budget is allocated for the following reservation period based on these predictions. These steps are illustrated in Figure 19.

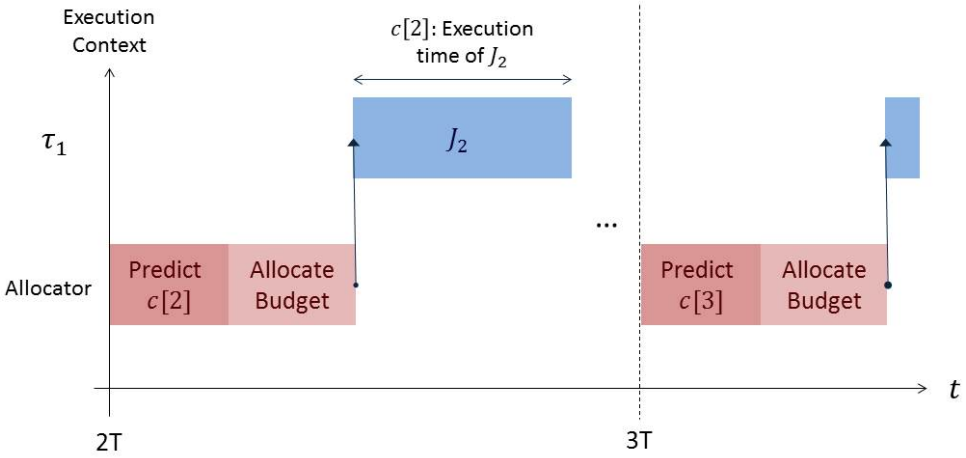


Figure 19: Budget allocation in the implementation of PBS. Job-execution times consist of only workload-specific computation.

In the implementation of TSP, SRT jobs consist of both workload-specific computation as well as prediction-related computation. In each SRT task, after the workload-specific computation completes, the execution time of the completed job is obtained from the kernel and used to predict the execution time of the following job. The predicted execution time is passed from the SRT task to the kernel module before the task is put to sleep. The kernel module stores the predicted values in shared memory pages that are accessible to the Allocator. When the Allocator is activated, the budget allocation for each SRT task is computed based on the corresponding

predicted values. These steps are shown in the example in Figure 20.

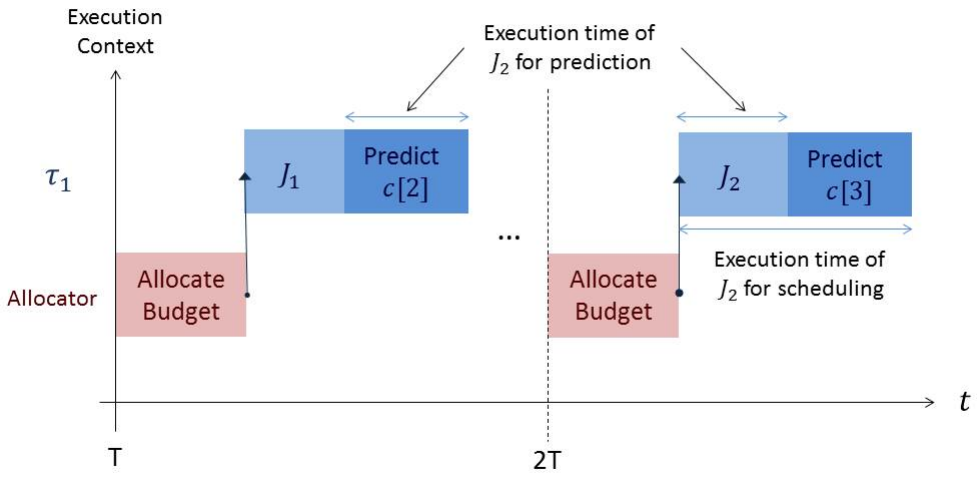


Figure 20: Budget allocation in the implementation of TSP. Execution-time prediction is done in the context of the corresponding SRT task. Job-execution time consists of both workload-specific computation and prediction-related computation.

Predicting the execution time of the next job, $c[j]$, requires knowledge of the execution time of the current job, $c[j-1]$. Also, prediction-related computation requires CPU time as well, and this CPU time is not known until the prediction operation completes. Therefore, for the purposes of prediction, the execution time of a job, J_j, is measured from the start of the prediction operation in the preceding job, J_{j-1}, until the end of the workload-specific computation in the current job, J_j. In this way, the execution time of J_j is available at the start of the prediction operation for J_{j+1}.

Consider the example in Figure 20. The execution time of J_2 consists of the prediction operation at the end of J_1 and workload-specific computation in J_2 and excludes the prediction operation at the end of J_2. In this way, the actual execution time of J_2 is available before predicting the execution time of J_3.

The above definition of job boundary and execution-time is appropriate for the purposes of prediction. However, for budget allocation, $c_{min}[k]$ is still measured such

that a job starts with the workload-specific computation and ends with the prediction operation. This definition of job boundary is more consistent with the periodic real-time task model where each job is released at the start of a task period and all parts of the job must complete within the task period. This difference in the definition of job boundary and the way that execution time is measured is not important because the execution time of the prediction-related computation remains mostly constant across jobs. Based on results presented in Section 4.5.4, prediction overhead has very little variability even across workloads. The differences in the way execution time is measured should not affect the analysis presented in Sections 4.2 and 4.3.

4.4.2 Soft-Real-Time Tasks

A basic template of an SRT tasks is presented in Figure 21. The interaction between SRT tasks and the kernel module is abstracted through a library called `libSRT`. After application-specific initialization, the predictor is setup and the `SRT_setup` function is called. In this function, the real-time scheduling policy and priority are configured and initial registration and setup is performed with the kernel module.

Before entering the main job loop, the `SRT_sleepTillFirstJob` function is called. In this function, the initial estimated job-execution time is passed to the kernel module and the SRT task is put to sleep until the next earliest reservation period. This initial sleep is done to align all task periods with system-wide reservation periods. Before the SRT task is scheduled, the Allocator is activated and the initial budget is allocated to the SRT task. When the SRT task is next scheduled, budget enforcement begins.

After the initial sleep, an SRT task enters the job loop where each loop iteration corresponds to a job. After any application-specific computation, the `SRT_sleepTillNextJob` function is called at the end of each job. For the purposes of prediction, this function call demarcates job boundaries. In this function, the execution time of the completed job is read from the kernel module and passed to the predictor. The predictor state

is updated and various prediction-related parameters are returned and passed to the kernel module. Depending on whether or not the deadline is missed, the SRT task is either allowed to continue to the next job or blocked until the following task period. On exiting the job loop, the `SRT_close` function is called which issues a command to the kernel module to stop the periodic activations, budget allocations, and budget enforcement.

```
/* Perform application-specific initialization */
...
/* Setup the predictor */
...
/* Initialize the CPU-budget mechanism */
SRT_setup( ... );

/* Notify the library of the start of the first job
*/
SRT_sleepTillFirstJob( ... );
while there are jobs remaining do
    /* Perform application-specific computation*/
    ...
    /* Notify the library that the current job is com-
    plete */
    SRT_sleepTillNextJob( ... );
end
/* Notify the library that all jobs are complete */
SRT_close( ... );

/* Perform application-specific cleanup */
...
```

Figure 21: Template of an SRT task.

4.4.3 Computing Budget Allocations

A separate predictor is maintained in each SRT task. The predictor is defined by a generic data structure that consists of a pointer to the predictor state and pointers to functions that define the operations of the predictor. When a job completes in an SRT task, the execution time of the completed job is read from the kernel module and passed to the `update` function of the predictor. In this function, the predictor

state is updated and four values are returned: `mu_c0`, `var_c0`, `mu_cl`, and `var_cl` as defined in Figure 22.

$$\begin{aligned}
\texttt{mu_c0} &= (\hat{c}_0[j] + \hat{\mathbb{E}}(e_0[j])) \\
\texttt{var_c0} &= \hat{\mathrm{Var}}(e_0[j]) \\
\texttt{mu_cl} &= (\hat{c}_l[j] + \hat{\mathbb{E}}(e_l[j])) \\
\texttt{var_cl} &= \hat{\mathrm{Var}}(e_l[j])
\end{aligned}$$

Figure 22: List of values required from a predictor for budget allocation. The variables, $\hat{c}_0[j]$, $e_0[j]$, $\hat{c}_l[j]$, and $e_l[j]$ are the single-step and multi-step predictions and prediction errors as defined in Figure 17.

On return from the `update` function, these four values and the `alpha` parameter are passed to the kernel module. The `alpha` parameter is maintained as a 32-bit fixed-point number with 16 fractional bits. The kernel module stores these values in a data structure called `SRT_loaddata`. A separate `SRT_loaddata` structure is maintained for each SRT task. In addition to the four prediction values and the `alpha` parameter, the `SRT_loaddata` structure contains the current job-queue length, the current run time, $c_{min}[k]$, and the number of reservation periods remaining until the next deadline. These additional fields are updated and maintained by the kernel module.

At each reservation-period boundary, the Allocator is activated and the data in each `SRT_loaddata` structure is used to compute the budget allocation for the corresponding SRT task. To reduce communication overhead, the array of `SRT_loaddata` structures is maintained in a set of memory pages that are shared between the kernel and the Allocator. Based on the equations presented in Figure 18, the budget allocation for each SRT task is computed as shown in Figure 23.

Budget allocations computed by the Allocator are written to memory pages that are shared with the kernel, similar to the pages that contain the array of `SRT_loaddata` structures. Once the budget allocation is computed for all SRT tasks, control returns to the kernel and the Allocator is put to sleep until the following reservation period. When control returns to the kernel, the kernel module checks the feasibility of

```
Result: budget
Data: mu_c0, var_c0, mu_cl, var_cl, current_runtime,
      queue_length, RPs_remaining

/*Check if the computational load is zero*/
if queue_length = 0 then
    estimated_load = 0;
else
    /*Compute the conditional mean minus the run-
    time*/
    c0_std = sqrt(var_c0);
    c0_load = mu_c0 - current_runtime;
    if c0_load < c0_std then
        c0_load = c0_std;
    end

    /*Compute the estimated mean and std*/
    total_mean = c0_load + (queue_length - 1)*mu_cl;
    total_var = var_c0 + (queue_length - 1)*var_cl;
    total_std = sqrt(total_var);

    /*Compute the estimated load*/
    estimated_load = total_mean + (alpha * total_std);
end
budget = estimated_load / RPs_remaining;
```

Figure 23: Algorithm for computing budget allocations for SRT tasks.

the budget allocations. In the event of an overload-condition, budget allocations are scaled back as described in Section 3.1.1.

4.5 Experiments and Results

4.5.1 Workloads and Experimental Platform

The LMS-MA Hybrid predictor and TSP budget-allocation algorithm have been tested with a range of multimedia workloads. Before presenting the results, the workloads and platform are described in greater detail.

All experimnets described in this section were performed on the platform described in Table 2. The custom-compiled kernel was configured to be non-SMP and with real-time group scheduling disabled. The CPU-frequency governor was set to "userspace"

and the CPU frequency was set to a constant value.

Table 2: Test Platform Configuration.

Processor	Intel Core 2 Quad
Clock Speed	2.403GHz
L1 D-cache	32kB
L1 I-cache	32kB
L2 Cache	4096kB
Operating System	Centos 6.4
Kernel	Linux 3.1.1

The application implementing the SRT multimedia workloads is based on the ffmpeg library [18]. Each job in this application consists of encoding or decoding a single frame of audio or video data. A number of steps are taken to to ensure that each job consists of only computation and memory operations and as a result, never blocks due to an I/O operation. For example, all the input data is read from multimedia files during initialization. The data is locked in memory using the `mlock` function to prevent paging. Also, the decoded or encoded frames are not sent to an I/O buffer or written to a file. It is important to prevent jobs from blocking on I/O operations to stay consistent with the model of periodic real-time tasks described in Section 2.2. The focus of the work presented here is on CPU-budget allocation and execution time prediction. For the purposes of this test application, the absence of I/O operations is acceptable. In more complete systems, I/O operations can be performed by separate hard-real-time threads using a pipelined software architecture as discussed in Section 3.1.

The multimedia workloads consist of encoding and decoding audio, speech, and video data. The speech encoding workload consists of encoding speech data into the AMR format. The speech decoding workload consists of decoding speech data from the speex format. The audio workloads consist of encoding and decoding audio data to and from the mp3 format. Lastly, the video workloads consist of encoding and

decoding raw video frames to and from the H.264 format.

4.5.2 Execution-time Characteristics

For each multimedia workload, the application was run on the platform described in Table 2 and job-execution times were measured. The mean, variance, and maximum of the job-execution times are presented in Table 3. Also, the table contains the magnitude and lag of the autocorrelation coefficient at its maximum. The autocorrelation coefficient is the autocorrelation function (ACF) normalized by the mean and variance. The heights of peaks in the ACF reflect the strength of any periodicity in the signal and the locations of these peaks reflect the period. The ACF has a significant effect on the prediction accuracy of LMS filters as shown in Section 4.5.4.

Table 3: Workload Execution-Time Statistics

Workload	Mean (ns)	Std (ns)	Max (ns)	Max ACF	Max ACF Lag
encode speech	3.35e+05	4.97e+04	5.19e+05	0.95	80
decode speech	7.16e+04	4.10e+03	3.53e+05	0.20	2
encode audio	1.01e+06	3.63e+05	2.36e+06	0.96	9
decode audio	9.65e+04	2.03e+03	1.83e+05	0.27	73
encode video	1.83e+07	5.00e+06	4.81e+07	0.52	2
decode video	1.15e+07	3.42e+06	2.97e+07	0.85	4

Execution-time statistics were collected from other multimedia workloads as well with different encoding formats and media files. In all workloads considered, the maximum execution time is significantly higher than the mean and in some cases, more than double. Also, in all cases, the standard-deviation of execution times is significantly smaller than the mean, often by more than an order of magnitude. The strength of the correlation between the execution times of successive jobs is affected by the encoding format and in some cases, the content of the data as well.

4.5.3 Predictor Configurations

In addition to the LMS-MA Hybrid predictor, two additional predictors were implemented for comparison of accuracy and overhead: a fixed-length moving average similar to the one used in the PBS algorithm and an array of moving averages as described in Section 4.2.1.

Every predictor implementation consists of a set of functions and state as defined in Section 4.4.3. A key function that defines the predictor is the `update` function. This function accepts the most recent job-execution time as input and computes predicted values and prediction-error statistics as outputs as defined in Figure 22.

Like the moving average used to implement the PBS algorithm, the fixed-length moving average consists of 20 taps. Both the predictor outputs `mu_c0` and `mu_cl` are set to the output of the moving average. The predictor outputs, `var_c0` and `var_cl` are set to the sample variance of the values in the moving window. For the purposes of numerical stability, variance is computed using the algorithm described in [46].

The array of moving averages consists of 7 filters with lengths that range from 2 to 128 in powers of two. The outputs `mu_c0` and `mu_cl` are set to the output of the least-MSE filter plus the estimated mean of the corresponding prediction error. The predictor outputs `var_c0` and `var_cl` are set to the estimated variance of the prediction error of the least-MSE filter.

Lastly, the LMS-MA Hybrid predictor consists of an array of 7 moving averages similar to the ones described above and five additional adaptive filters. The five adaptive filters are based on the normalized variable-step LMS algorithm described in Section 4.2.2, and have lengths that range from 2 to 32 in powers of two. While longer adaptive filters can exploit correlation at larger lags, such filters are slower to converge and have higher overheads. Therefore, the length of the adaptive filters are limited to 32. As in the case of the array of moving averages, the predictor outputs `mu_c0` and `mu_cl` are set to the output of the least-MSE filter plus the estimated mean

of the corresponding prediction error. The predictor outputs `var_c0` and `var_cl` are set to the estimated variance of the prediction error of the least-MSE filter.

4.5.4 Predictor Accuracy and Overhead

The accuracy and overhead of the first-stage prediction algorithm affects the overall performance of the TSP algorithm. The execution time and normalized RMS prediction error of the predictors described above were measured for the platform and multimedia workloads described in Section 4.5.3.

In Table 4, the average overhead associated with each of the three algorithms are shown. The overhead is consistent across the different workloads. At hundreds of nano seconds, these overheads are orders of magnitude smaller than the mean job-execution times shown in Table 3. As expected, LMS-MA Hybrid has the highest overhead of the three algorithms. On the other hand, the array of moving averages has a lower overhead than a simple fixed-length moving-average. The higher overhead for the fixed-length moving average may be associated with the algorithm used to compute variance. In contrast, the EWMA used in the array of moving averages incurs much less overhead.

Table 4: Comparison of Predictor Overheads (in ns)

Workload	MA	MA Array	LMS-MA Hybrid
encode speech	302.32	222.68	635.35
decode speech	302.98	230.04	650.41
encode audio	301.67	222.91	629.31
decode audio	303.14	226.18	646.05
encode video	304.83	229.01	649.32
decode video	300.12	225.00	635.85

In Table 5, the RMS error is shown for each of the prediction algorithms and

workloads. The RMS error values are normalized by the standard deviation of job-execution times of the corresponding workloads. The standard deviation of job-execution times is in thoery the lowest RMS error possible using a static predicted value. Based on results shown in the table and results from additional workloads, the

Table 5: Comparison of Normalized RMS Prediction Error

Workload	MA	MA Bank	LMS Hybrid
encode speech	1.04	1.03	0.74
decode speech	1.03	1.15	1.11
encode audio	1.01	1.01	0.64
decode audio	0.86	1.27	0.83
encode video	0.96	0.97	1.62
decode video	0.66	0.62	0.45

LMS-MA Hybrid algorithm has the least RMS prediction error for most multimedia workloads where there is a strong correlation in the execution times of successive jobs. The exception is webm video encoding and mpeg4 video encoding including the video-encoding workload referenced in Table 5. The reason for the high RMS errors with these workloads is not clear. The LMS-MA Hybrid algorithm also works well with certain workloads with weak correlation in execution times. However, for most workloads with weak correlation, the moving-average-based algorithms perform better.

4.5.5 Performance of the TSP Algorithm

The implementation of the TSP algorithm was tested with the multimedia workloads and the three first-stage predictors described above. For each workload and predictor, experiments were performed with different values of alpha. The values of alpha were chosen such that the deadline miss rate ranged between 2% and 20%. Equation 66 for computing alpha is based on Cantelli's inequality which is conservative. Therefore, the alpha parameter had to be tuned such that the resulting deadline-miss rate was

in the desired range.

For each workload, prediction algorithm, and value of alpha, the experiment was repeated five times. For each repetition of an experiment, the performance was measured over the duration of the workload in terms of deadline-miss rate, normalized budget allocation, and normalized RMS VFT error.

Normalized budget allocation is the average per-job budget allocation divided by the mean job-execution time. The mean job-execution time is equivalent to the mean per-job CPU-budget consumption. The normalizing factor, the mean job-execution time, does not change with the prediction algorithm beyond differences in the prediction overhead, and the prediction overhead is negligible compared to the mean job-execution time. Normalized budget allocation represents the factor of budget over allocation. The ideal value of normalized budget allocation is one, and for reasons of queue stability, the normalized budget allocation is usually more than one. Generally, the average budget allocation can be reduced at the cost of an increase in the deadline-miss rate by reducing the value of alpha. It is desirable to have both a low average budget allocation and a low deadline-miss rate.

The *RMS VFT error* is the root-mean-square difference between virtual finishing times and job deadlines as defined in Section 3.1.2. The VFT error is large and positive when a deadline is missed by a large lag, and large and negative when the CPU budget is over allocated by a large amount. The RMS VFT error is large when either of these cases occur frequently. Therefore, it is desirable to have a low RMS VFT error. The RMS VFT error is normalized by the task period.

In the remainder of this section, results are presented for each workload in two scatter diagrams of the deadline-miss rate (x-axis) against normalized budget allocation and normalized RMS VFT error, respectively. Since it is desirable to reduce all three of these metrics, points that are closer to the axes represent better performance. As expected, the overall performance of the TSP algorithm generally depends on the

accuracy of the first-stage prediction algorithm. For a given deadline-miss rate, lower RMS prediction error usually results in lower budget allocation and lower RMS VFT error.

Results for the speech decoding workload are shown in Figure 24. From Table 5, the normalized RMS prediction error associated with the MA, MA Array, and LMS-MA Hybrid predictors are 1.03, 1.15, and 1.11, respectively. Accordingly, when MA is used as the first-stage predictor, the RMS VFT error and the normalized budget allocation is the lowest for any given miss rate. When LMS-MA Hybrid is used, the performance is worse than when MA is used, but better than when MA Array is used.

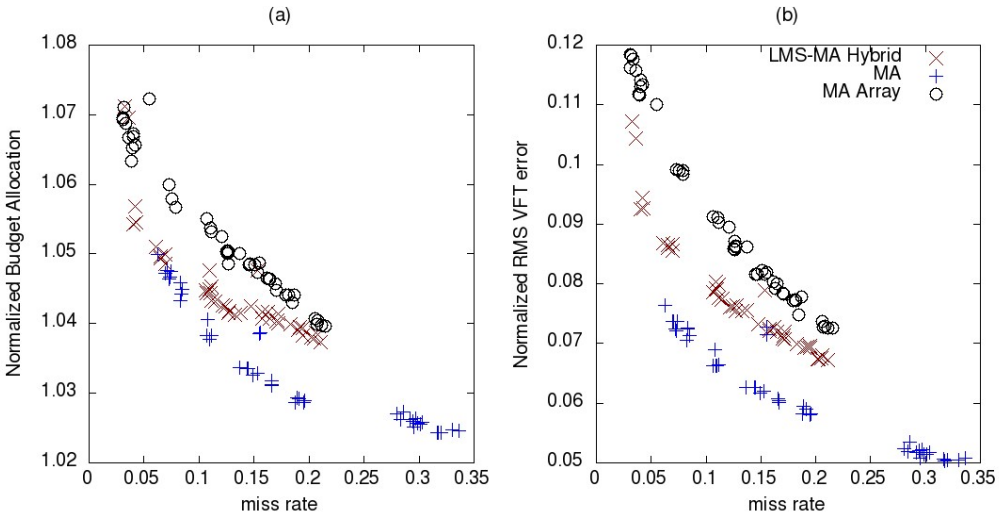


Figure 24: Performance of the TSP algorithm with a **speech-decoding** workload.

Results corresponding to the speech encoding workload are shown in Figure 25. For this workload, the normalized RMS prediction error of the MA, MA Array, and LMS-MA Hybrid predictors are 1.04, 1.03, and 0.74, respectively. The RMS VFT error values are consistent with the RMS values of the prediction error. Specifically, the LMS-MA Hybrid algorithm has the least RMS prediction error and also has the least RMS VFT error for any given deadline-miss rate. However, for the MA and MA Array algorithms, the normalized budget allocation decreases after peaking at about

1.33 even as the value of alpha increases and the deadline-miss rate decreases.

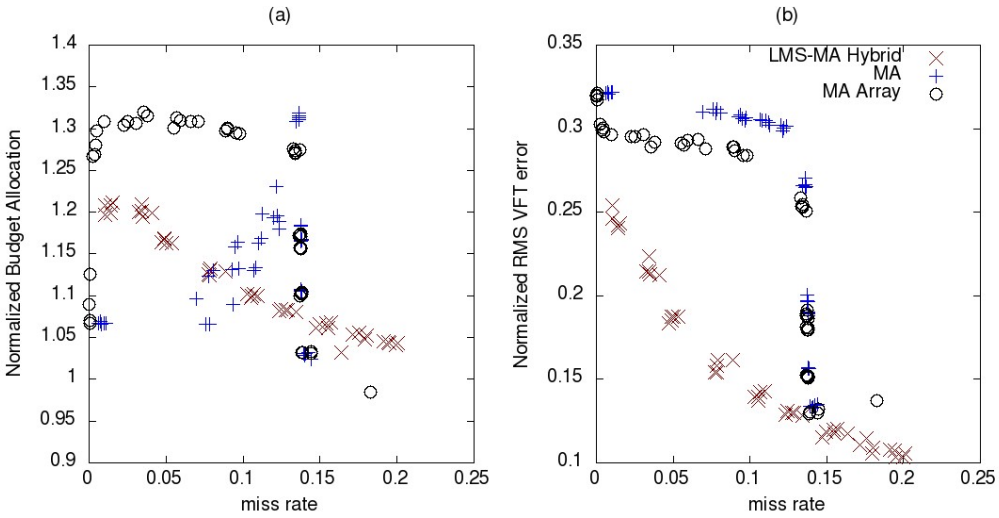


Figure 25: Performance of the TSP algorithm with a **speech-encoding** workload.

This behavior pertains to the way TSP works in the case of budget over allocation. When the normalized budget allocation is equal to (4/3), the average budget allocation is (4/3) times the average job-execution time. In this case, the length of the task period is four times the length of the reservation period and therefore, jobs complete by the end of the third reservation period. Furthermore, budget allocation is adapted at each reservation-period boundary. When it is detected at the start of the fourth reservation period that the job completed early, no budget is allocated. As a result, the average budget allocation over the four reservation period roughly equals the average job-execution time. As a metric of performance, there is no penalty in average budget allocation for early job completions in the way that there is in RMS VFT error.

Results corresponding to the audio decoding workload and video decoding workload are shown in Figure 26 and Figure 27, respectively. As in the case of the speech workloads, the performance of the TSP algorithm is closely related to the RMS prediction error of the first-stage predictor.

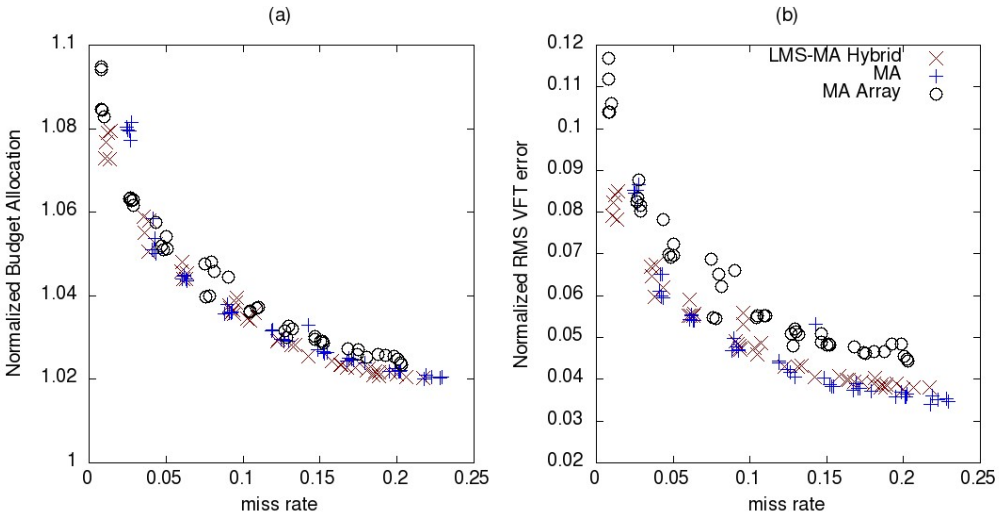


Figure 26: Performance of the TSP algorithm with an **audio-decoding** workload.

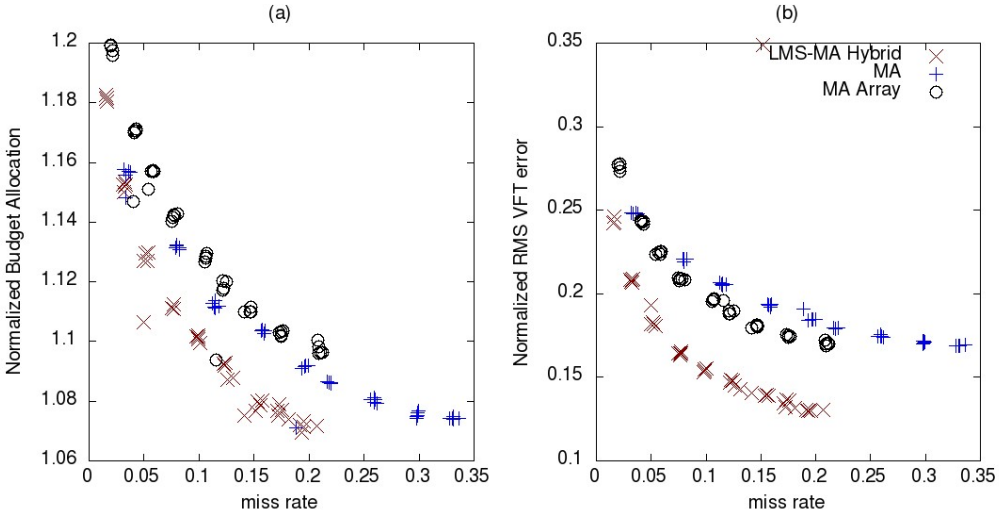


Figure 27: Performance of the TSP algorithm with a **video-decoding** workload.

Results corresponding to the audio encoding workload and video encoding workload are shown in Figure 28 and Figure 29, respectively. Unlike the case of the previous workloads, RMS VFT error and normalized budget allocation are not consistent with the RMS error of the first-stage predictor. Specifically, the LMS-MA Hybrid algorithm has the lowest RMS prediction error for the video-encoding workload but incurs the largest RMS VFT error for a given miss rate. On the other hand, the LMS-MA Hybrid algorithm has the highest RMS prediction error for the audio-encoding workload, but performs similarly to the MA and MA Array algorithms. The

reason for the inconsistency with these two workloads is not clear.

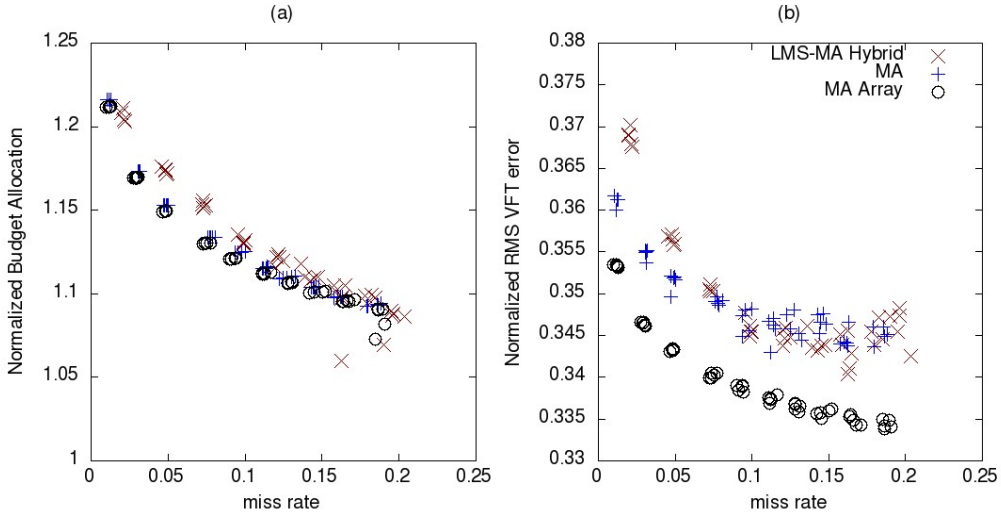


Figure 28: Performance of the TSP algorithm with an **audio-encoding** workload.

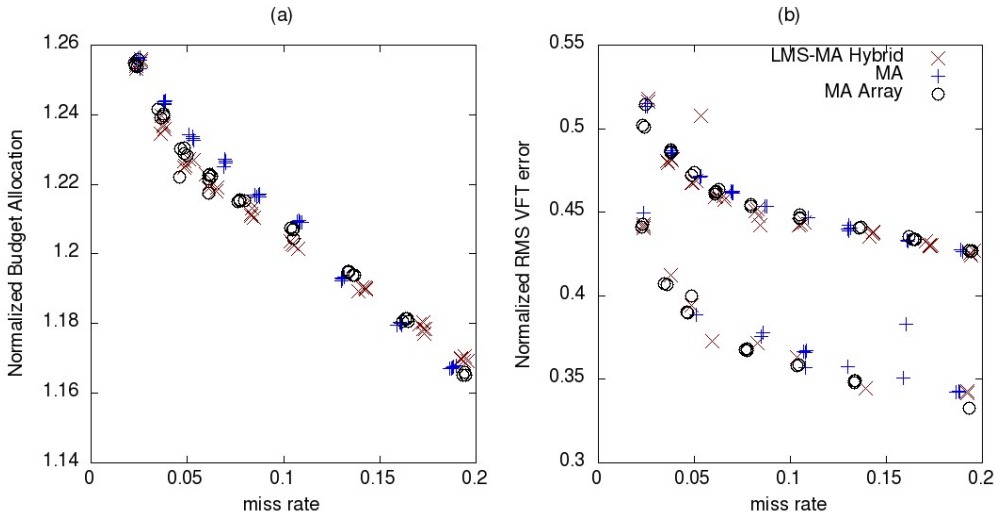


Figure 29: Performance of the TSP algorithm with a **video-encoding** workload.

4.6 Conclusion

Overall, the TSP algorithm improves asynchronous budget allocation in a number of ways. The second stage of the TSP algorithm guarantees a bound on the probability of a job missing the deadline. This is a stronger guarantee than the bounded-queue-length guarantee provided by the algorithm in Chapter 3. Furthermore, this guarantee is provided without the i.i.d assumption made in Chapter 3.

There is a strong correlation in job-execution times for some multimedia workloads. Instead of assuming that job-execution times are i.i.d, the TSP algorithm allows correlation in job-execution times to be exploited to make better predictions. In cases where execution times are correlated, the LMS-MA hybrid algorithm is a more accurate predictor of execution times than a simple moving average. With more accurate predictions of execution times from the first-stage predictor, the overall performance of the TSP algorithm is better in most cases.

The TSP algorithm is flexible in the choice of the first-stage predictor. The software architecture presented earlier in the chapter, allows for the use of an application-specific first-stage predictor. If such a predictor offers more accurate predictions of job-execution times than the moving average or LMS-MA hybrid algorithm, then the performance of the TSP algorithm may be improved even further.

CHAPTER V

VIC-BASED BUDGET ALLOCATION

Managing the power consumption of a platform involves configuring the platform to the appropriate mode of operation. Different modes of operation such as CPU frequency settings offer different levels of trade off between performance and power dissipation.

Therefore, power-management decisions in real-time systems should be based on CPU-budget allocations. Specifically, the platform should not be configured to such a low speed that CPU-budget commitments to real-time tasks can no longer be satisfied. At the same time, the platform should not be configured to a speed that is greater than what is necessary to satisfy budget commitments if the platform requires additional energy to run at that higher speed.

On the other hand, changes in the platform performance results in changes in execution times. So, when the mode of operation is changed for the purposes of power management, CPU-time allocations must be modified to handle the resulting changes in job-execution times. For example, if the CPU speed is reduced, CPU time allocations must be increased to accommodate longer job-execution times. Power-management decisions and CPU-time allocations are interdependent. These decisions can be made simpler if allocation decisions can be made independent of power-management decisions.

While CPU-budget mechanisms presented in most previous articles are based on the allocation of CPU time, the CPU-budget concept can be generalized to other measures of computation as well. For example, CPU-budget allocations in the GRACE-OS systems are specified in terms of CPU cycles[72]. If the measure of computation

used for budget allocation is invariant under change in the mode of operation, then the mode of operation is no longer relevant for the purposes of budget allocation and budget-allocation decisions become independent of power-management decisions.

A measure of computation is said to be invariant under change in the mode of operation if based on this measure, the amount of computation needed to perform a job is the same regardless of the mode of operation. For example, for CPU-bound workloads, the number of CPU cycles required to perform a job is the same regardless of CPU frequency. For both CPU-bound and memory-bound workloads, the number of user-level instructions that must be retired to perform a job is the same regardless of the CPU frequency. Along the same lines, a novel measure of computation is presented in this chapter, called *Virtual Instruction Count* (VIC). VIC is an approximation of retired-instruction count based on dynamic estimates of the average instruction-retirement rate. However, unlike retired-instruction count, VIC increases at a known rate over time, a property that is needed for budget allocation.

A VIC-based budget mechanism has been implemented in Linux by extending the TSP implementation described in Section 4.4. This implementation has been tested with synthetic and video-decoding workloads. Similar tests were performed on the time-based budget mechanism described in Chapter 4. Results from these experiments show that VIC-based budget allocation allows for a faster response to changes in the CPU frequency than time-based-budget allocation.

The remainder of this chapter is organized as follows. In Section 5.1, experimental results are presented to motivate the need for a novel measure of computation for budget allocation. Specifically, experimental results are presented to demonstrate that CPU time, CPU cycles, and retired-instruction count are not appropriate measures of computation for CPU-budget allocation in real-time systems. Then, in Section 5.2, a more precise definition of VIC is presented, VIC-based budget is described in greater detail, and the adaptive performance model is presented. Examples are presented

to illustrate VIC and VIC-based budget allocation. In Section 5.3, implementation details are presented for a VIC-based budget mechanism in Linux. Finally, in Section 5.4, experimental results are presented to demonstrate the invariance property of VIC, and to demonstrate the advantages of VIC-based budget allocation over time-based budget allocation.

5.1 The Need for an Alternative Measure of Computation

On real-time systems, the actual resource that is allocated to tasks is time. Therefore, when some amount of CPU capacity is committed to the hosted task set over a reservation period, it should be possible to guarantee that the committed capacity is delivered within the reservation period. If this guarantee cannot be provided, either the the actual allocated capacity may be less than the committed capacity, or timing constraints may be violated.

For example, consider the case where the instruction-retirement rate is mispredicted, 10^6 instructions are committed to a task set over a $10ms$ reservation period, and the actual average instruction-retirement rate is 9×10^8 instructions per second. The actual number of instructions retired over the reservation period is 9×10^5. In this case, either the task set will be allocated only 9×10^5 instructions of the 10^6 committed instructions, or the task set must over run into the following reservation period by $1.1ms$ to allow the full 10^6 instructions to retire. On real-time systems, neither of these possibilities is acceptable. Therefore, *the measure of computation used for budget allocation on a real-time system should increase over the reservation period at a rate that is known at the start of the reservation period.*

In addition, as mentioned at the beginning of this chapter, *it is desirable to use a measure of computation for CPU-budget allocation that is invariant under change in the mode of operation.* Specifically, it is desirable to to use a measure of computation for which the CPU-budget required to complete a job remains the same regardless of

the mode of operation. This property of invariance can help to reduce or eliminate transient behavior in the adapted budget allocation that results from changes in the mode of operation. While this invariance property is desirable, it is not a strict requirement for soft-real-time systems.

In the remainder of this section, experimental results are presented to demonstrate that certain measures of computation are not appropriate for the purposes of CPU-budget allocation and to motivate the need for a novel measure of computation. Specifically, results are presented to demonstrate that the number of CPU-clock cycles required to execute a job is not always invariant under change in the CPU frequency. Also, results are presented to show that job-execution times are not necessarily inversely proportional to the CPU frequency.

Further results are presented to show that the time required to retire a specific number of instructions can vary widely across workloads, and that the instruction-retirement rate for the same workload can vary with time. Because of this uncertainty in instruction-retirement rate, retired-instruction count is not an appropriate measure of computation for budget allocation.

All experiments described in this chapter were performed on the platform described in Table 6. A custom-compiled kernel was used that was configured to be non-SMP, and real-time group scheduling was disabled. The CPU frequency was controlled through the `cpufreq` subsystem[40]. The CPU governor was set to `userspace`, and the CPU frequency was set as needed.

5.1.1 A Synthetic Workload to Control Memory Boundedness

The number of cycles required to execute a job can change due to off-chip components of execution time. Examples of off-chip components of execution time include the time spent waiting for the completion of I/O operations or more commonly, for the completion of memory operations. Because this time is unaffected by the CPU-clock

Table 6: Test Platform Configuration.

Processor	Intel Core i7-3610QM
L1 D-cache	32kB
L1 I-cache	32kB
L2 Cache	256kB
L3 Cache	6144kB
Operating System	Ubuntu 12.04
Kernel	Linux 3.2.0

speed, more cycles are wasted waiting for the completion of the same operations when the CPU frequency is higher.

To demonstrate the effects of memory boundedness, a synthetic workload has been created called *membound*. This workload is based on the traversal of a circular linked list. Each job in this workload consists of a fixed number of steps through the linked list and the number of steps is used to control the job-execution time.

By nature of linked lists, the memory address of the next node is not known until the previous node is accessed. When the list is traversed, this dependence between adjacent memory accesses prevents these memory operations from being parallelized by the hardware. Furthermore, adjacent nodes in the list are arranged randomly over a memory region of fixed size. This random arrangement is meant to eliminate any stride patterns in the memory accesses that could by exploited by a hardware prefetcher. These steps are taken to ensure that the workload spends a greater portion of the execution time waiting for memory operations to complete.

Each node in the linked list is padded and aligned to cache-line size. The number of nodes in the list is a configurable parameter used to control the working-set size of the workload. The working-set size is used to control how frequently the different cache levels are missed. When the working-set size is significantly larger than the last-level cache, there are frequent cache misses and the workload is memory bound. When the working-set size is sufficiently small, there are no cache misses after the

cache is warmed and the workload is chip-bound.

5.1.2 Invariance Under Change in CPU Frequency

To test for invariance under change in CPU frequency, experiments were performed on a number of different **workloads**, measuring the average per-job CPU usage based on different **measures of computation** at different **CPU frequencies**. The workloads used in this experiment include two different configurations of the membound workload described in Section 5.1.1 and an H.264 video-decoding workload.

In the first configuration of membound, called *cache line*, the linked list consists of a single node linked to itself, and the workload consists of repeatedly accessing the same cache line. In the second configuration of membound, called *thrash*, the linked list consists of enough nodes to span a memory region that is eight times the size of the last-level cache. In this latter configuration, the workload misses the last-level cache on every step through the linked list. In both of these configurations of membound, the number of steps through the list is kept the same for all jobs and the instructions executed are the same; only the data is different. The first configuration is designed to make the workload chip-bound, whereas the second configuration is designed to make the workload memory-bound. For the video-decoding workload, each job consists of decoding a single video frame from an H.264 encoded video file.

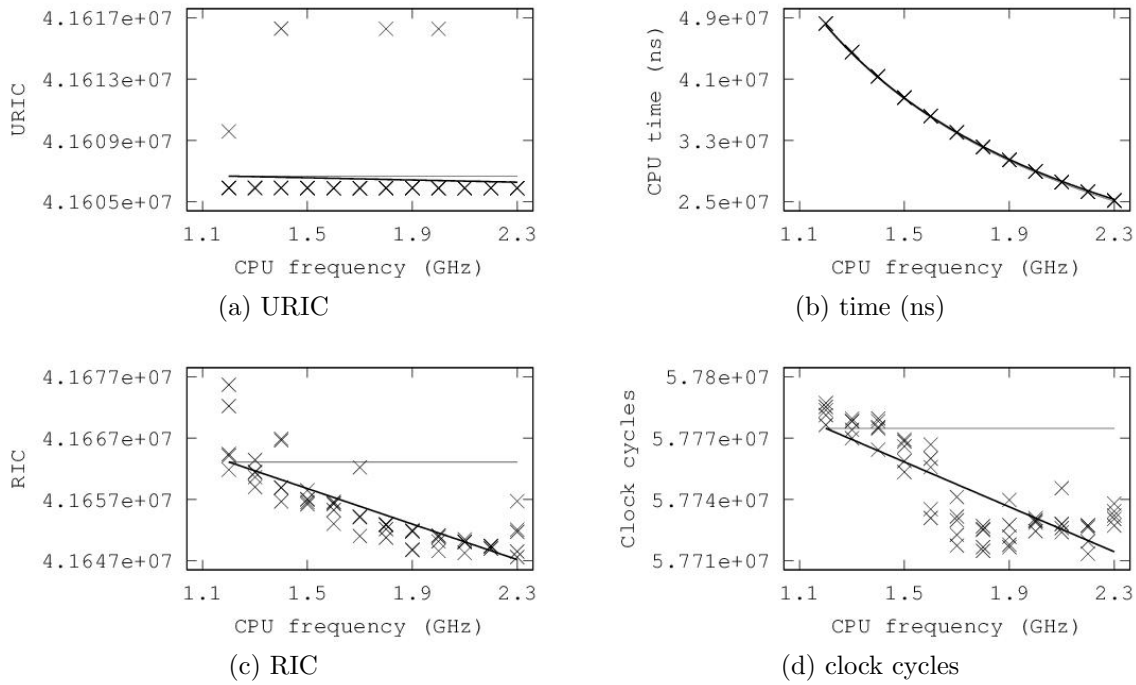
For each workload, the experiment was performed at different CPU frequencies. For each workload and frequency, CPU usage for each job was measured separately with different measures of computation: time, clock cycles, retired-instruction count (RIC), and user-level retired-instruction count (URIC). Other than time, measurements for all other measures were taken using hardware performance counters (PMCs). The PMCs were accessed through the `perf` interface in the Linux kernel. URIC and RIC differ in that URIC excludes instructions executed in kernel space handling interrupts and exceptions, whereas RIC includes all retired instructions.

For each workload, clock frequency, and measure of computation, the experiment was repeated five times. For each repetition, the measured CPU usage was averaged across all jobs. Results corresponding to the membound "cache line" workload, the membound "thrash" workload, and the H.264 video-decoding workload are shown in Figures 30, 31, and 32, respectively. In each figure, a separate plot is shown for each of the measures of computation. The horizontal axis corresponds to CPU frequency, the vertical axis corresponds to the average per-job CPU usage, and each marker corresponds to a single run. The thick black lines are best-fit curves, whereas the thin gray lines are curves expected from a chip-bound workload.

5.1.2.1 The membound "cache line" workload

The "cache line" configuration of the membound workload corresponds to an ideal chip-bound workload. As shown in Figure 30a, the average number of user-level instructions required to complete a job remains roughly the same across all CPU frequencies. This behavior is expected since a job is defined by the user-level instructions needed to complete the job. Also, as shown in Figure 30b, the average job-execution time decreases with increasing CPU frequency. As expected, the execution time is inversely proportional to CPU frequency.

However, the average per-job RIC decreases with increasing CPU frequency as shown in Figure 30c. The difference between RIC and URIC is the kernel-level RIC. When the CPU frequency is faster, the average execution time is smaller. As a result, fewer interrupts are handled during the execution of a job, and the RIC is smaller. With fewer instructions to retire, the average number of CPU cycles required to complete the jobs is smaller as well. As shown in Figure 30d, the average number of CPU cycles required to complete a job decreases with CPU frequency. This decrease in the number of instructions and cycles is small compared to the total number of instructions and cycles.



(a) URIC (b) time (ns)

(c) RIC (d) clock cycles

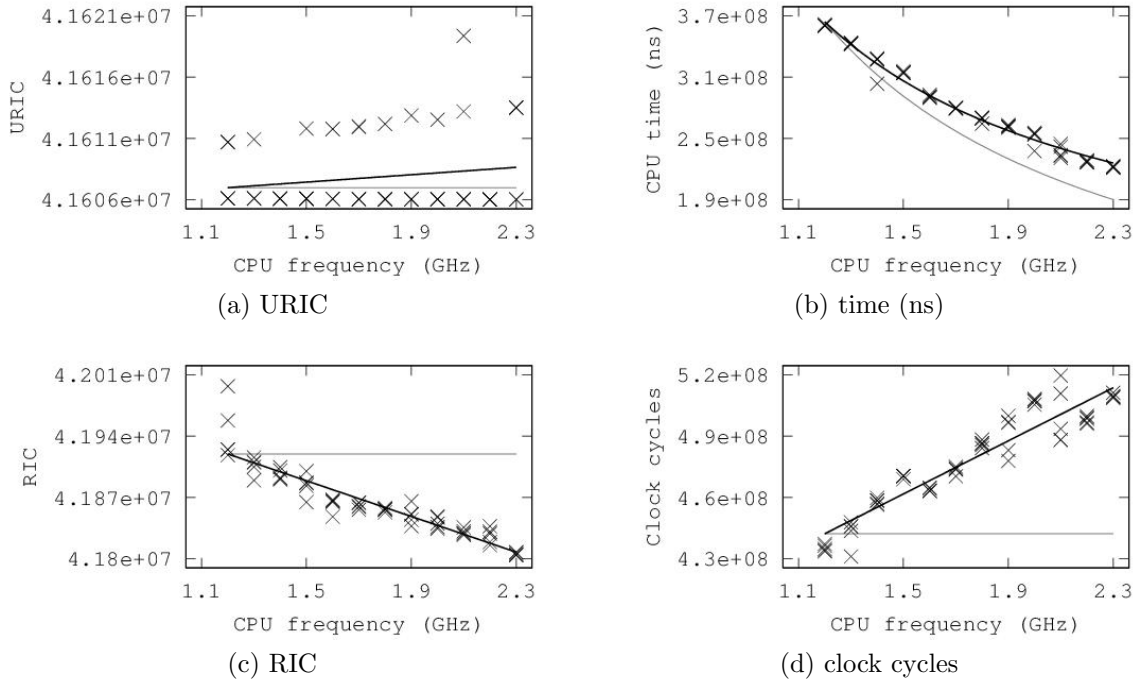
Figure 30: Average per-job CPU usage for the membound workload in the "cache line" configuration at different clock frequencies.

5.1.2.2 The membound "thrash" workload

Results for the "thrash" configuration of the membound workload are shown in Figure 31. The average number of user-level instructions required to complete a job is roughly the same across all CPU frequencies. The average job-execution time decreases with increasing CPU frequency. The retired-instruction count decreases with CPU frequency.

However, as shown in Figure 31d, the average number of CPU cycles required to complete a job increases significantly with CPU frequency. Since this workload is memory bound, a significant component of the execution time is spent waiting for off-chip memory operations to complete. This component of execution time does not decrease with an increase in CPU frequency. As a result, when the frequency is higher, more clock cycles are wasted waiting for the same operation to complete.

An effect of the off-chip component of execution time can be seen in Figure 31b



(a) URIC (b) time (ns)

(c) RIC (d) clock cycles

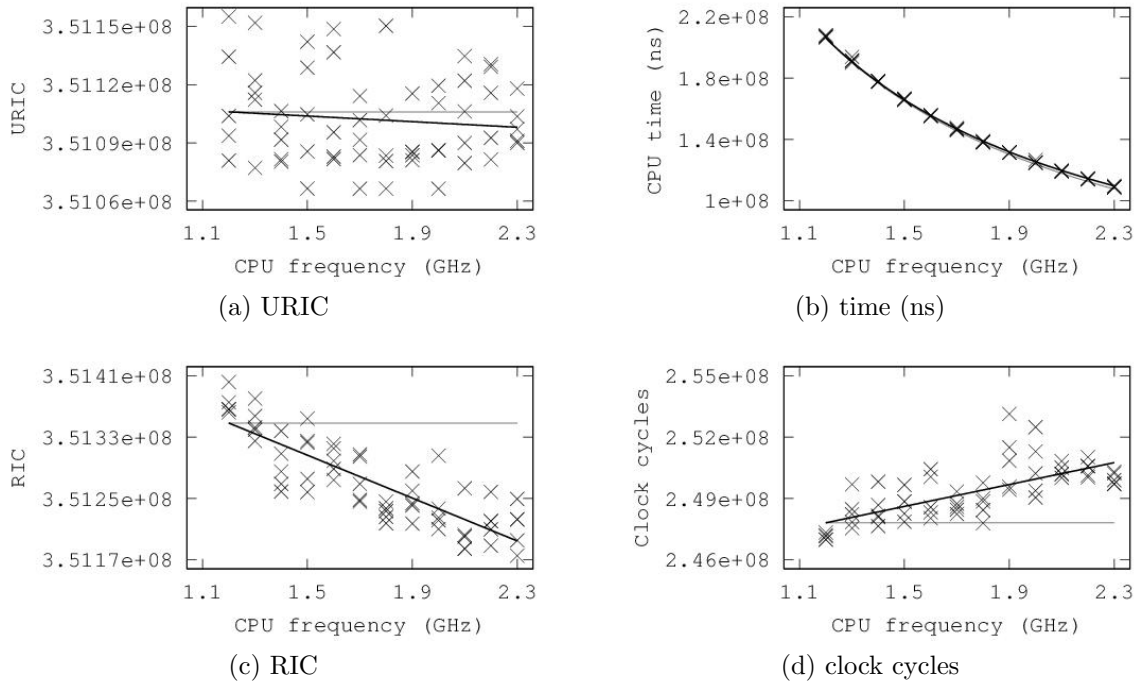
Figure 31: Average per-job CPU usage for the membound workload in the "thrash" configuration at different clock frequencies.

as well. The thin gray line represents the expected job-execution time at each CPU-clock frequency assuming that execution times are inversely proportional to frequency. However, the thick black line, representing the best-fit line, shows that actual execution times are higher than expected at higher frequencies. The execution times are higher because the off-chip component of execution time does not decrease with CPU frequency.

5.1.2.3 The Video-Decoding Workload

For the video-decoding workload, the relationship between average CPU usage and CPU frequency is the same as for the "thrash" configuration of the membound workload. The average number of user-level instructions required to complete a job still remains more or less the same across all CPU frequencies as shown in Figure 32a. The average job-execution time decreases with increasing CPU frequency as shown in Figure 32b. The retired-instruction count decreases with CPU frequency as shown

in Figure 32c. Lastly, the average number of CPU cycles required to complete a job increases with CPU frequency as shown in Figure 32d. However, actual job-execution times, as shown in Figure 32b, are not visibly larger than ideal job-execution times as in the membound "thrash" workload. These results suggest that the video-decoding workload is memory bound but not as severely as the membound "thrash" workload.



(a) URIC (b) time (ns)

(c) RIC (d) clock cycles

Figure 32: Average per-job CPU usage for the H.264 video-decoding workload at different clock frequencies.

5.1.3 Variability in Instruction-Retirement Rate

It is **desirable** that the measure of computation used for CPU-budget allocation should be invariant under change in the mode of operation. Based on the results in Section 5.1.2, user-level retired instruction count exhibits this property better than any of the other measures of computation. However, it is **required** that the measure of computation used for budget allocation on a real-time system should be deterministic. Specifically, the measure of computation should increase over a reservation period at a rate that is known at the start of the reservation period. In the case

of user-level retired instruction count, the rate of increase is the retirement rate. In this section, experimental results are presented to show that retirement rate can vary widely across workloads and can vary even within the same workload. Therefore, the retirement rate is difficult to predict accurately ahead of time.

5.1.3.1 Variability Across Workloads

To produce a range of instruction-retirement rates, the membound workload was run in three different configurations in addition to the ones described in Section 5.1.2. Specifically, the number of nodes in the linked list was configured such that the size of the list equals the size of the L1-data cache, the L2 cache, and the L3 cache, respectively. The sizes of the linked list in the different configurations are shown in Table 7. By controlling the working-set size of the workload, it is possible to control how frequently the different cache levels are missed, and to vary the instruction-retirement rate.

Table 7: The size of the linked list in the different configurations of the membound workload.

Configuration Name	Number of Nodes	Size in Memory
cache line	1	64B
L1-cache	512	32kB
L2-cache	4k	256kB
L3-cache	96k	6MB
thrash	768k	48MB

For each configuration, the user-level instruction count and execution time was measured per job and averaged across all jobs. The average instruction counts and execution times for each workload are shown in Figures 33a and 33b, respectively. For each job in each configuration, the number of steps taken through the linked list was kept the same, and so the average instruction count is the same across all configurations However, even though the instruction count is the same, the average

job-execution times from the "L1-cache" configuration and "thrash" configuration differ by an order of magnitude. The amount of time required to retire a specific number of instructions can differ by an order of magnitude across workloads.

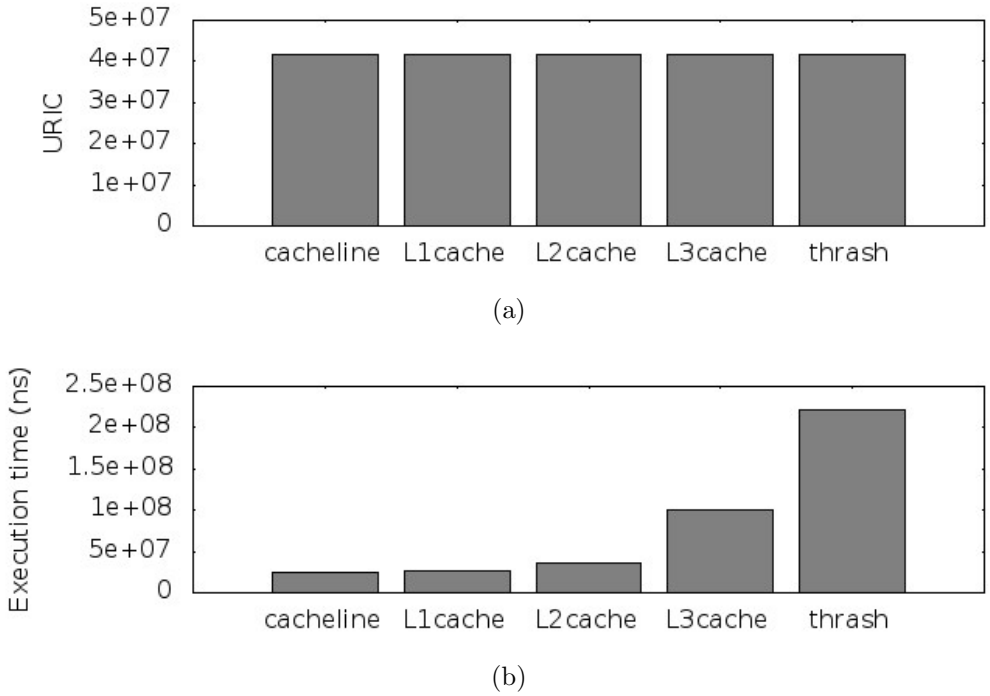


Figure 33: The average per-job URIC (top) and job-execution time (bottom) for different configurations of the membound workload described in Table 7.

5.1.3.2 Variability Within a Workload

Further experiments were performed to study how the instruction-retirement rate can vary over time within the same workload. The video-decoding workload described in Section 5.1.2 was run with the CPU clock set to a constant frequency. The `perf` interface was used to configure hardware performance counters to count user-level instructions and clock cycles. At the start of the workload, a high-resolution timer was setup to fire every $2.5ms$. When the timer fired, the relevant performance counter was sampled and logged with a time stamp. Once the workload was complete, the time stamps and counter values were used to compute the average rate of increase in clock cycles and user-level instructions over each $2.5ms$ interval. The plots of average

clock frequency against time and retirement rate against time are shown in Figures 34a and 34b, respectively. The clock frequency remains mostly constant with time as expected. The small variability in clock frequency can be attributed to noise in the time stamps.

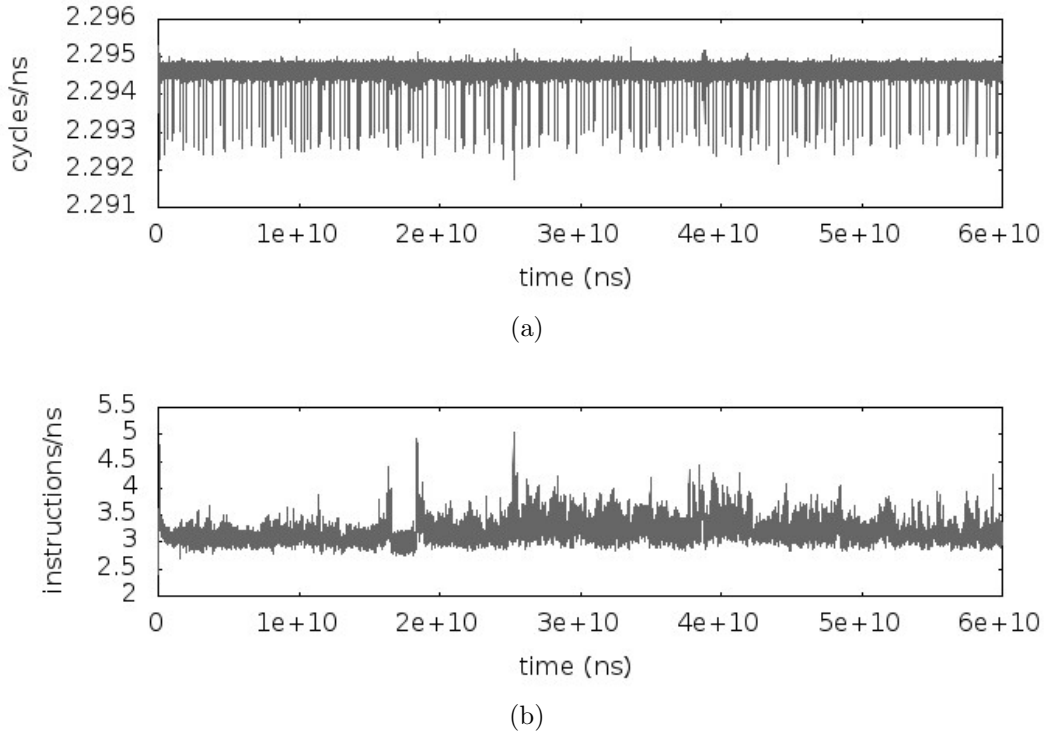


Figure 34: Average CPU frequency (top) and average user-level instruction-retirement rate (bottom) over time for a video-decoding workload.

In comparison, there is significant variability in the retirement rate. This variability is likely due to phase behavior in applications; most applications go through different phases of memory boundedness and CPU boundedness. Phase identification and prediction for the purposes of power management has been explored previous articles [42]. However, phase-based power management is beyond the scope of this dissertation.

5.2 Virtual Instruction Count

Retired-instruction count, while invariant under change in the mode of operation, is not an appropriate measure of computation for CPU-budget allocation because instruction-retirement rate is not predictable. An alternative to directly allocating CPU budget in terms of retired-instruction count is to allocate CPU-budget in terms of *Virtual Instruction Count* (VIC). VIC is an approximation of RIC based on dynamic estimates of the retirement rate. However, the rate of increase in VIC is known ahead of time over finite intervals of time, which makes reliable budget allocation in terms of VIC possible. Also, because VIC approximates RIC, and RIC is invariant under change in the mode of operation, VIC is also invariant to a degree that depends on the accuracy of the approximation.

5.2.1 Computing VIC

Let the platform modes of operation be indexed $1, 2, ..., M$ in order of increasing performance, such that M denotes the index of the highest-performance mode. The index of the active mode of operation at time t is a continuous-time discrete signal, denoted $m(t)$. A performance-model coefficient, $\hat{\mu}_m$, is associated with each mode of operation and represents an estimate of the average instruction-retirement rate in that mode. These estimates are updated at each reservation-period boundary and remain constant over the reservation period.

The performance-model coefficient for the m^{th} mode of operation in the k^{th} reservation period is denoted $\hat{\mu}_m[k]$. The expression, $(\lfloor t/T_R \rfloor)$, corresponds to the index of the reservation period at time t where T_R denotes the length of a reservation period. Since the index of the active mode of operation at time t is $m(t)$, the estimated average instruction-retirement rate at time t is $\hat{\mu}_{m(t)}\left[\lfloor t/T_R \rfloor\right]$. Based on this estimate, the VIC is defined as follows:

$$VIC(t) = \int_0^t \hat{\mu}_{m(\tau)}[\lfloor \tau/T_R \rfloor]\, \mathrm{d}\tau. \tag{85}$$

The estimated retirement rate, $\hat{\mu}_{m(t)}\left[\lfloor t_0/T_R \rfloor\right]$, can only change at two types of events: at reservation-period boundaries and at transitions in the mode of operation. Between these events, the estimate remains constant. Therefore, $VIC(t)$ is a piecewise-linear function of time. If a time, t_0, is chosen such that there are no mode transitions or reservation-period boundaries between t_0 and t, $VIC(t)$ can be expressed recursively as follows:

$$VIC(t) = VIC(t_0) + (t - t_0) \cdot \hat{\mu}_{m(t_0)}\left[\lfloor t_0/T_R \rfloor\right]. \tag{86}$$

For the purposes of implementation, it is more convenient to work with this recursive definition of VIC.

5.2.2 VIC Source

The implementation of a VIC-based budget mechanism requires a monotonically increasing VIC source similar to a clock source. An implementation of a VIC source based on (86) must maintain some state. This state consists of the last time stamp, t_0, the value of $VIC(t_0)$, and the estimated retirement rate at that time, $\hat{\mu}_{m(t_0)}\left[\lfloor t_0/T_R \rfloor\right]$. During initialization, t_0 should be set to the current time, $VIC(t_0)$ should be set to zero, and $\hat{\mu}_{m(t_0)}\left[\lfloor t_0/T_R \rfloor\right]$ can be set based on an initial estimate of the retirement rate for the active mode of operation. Then, this state must be updated immediately after every mode transition and reservation-period boundary. The new value of $VIC(t_0)$ can be computed using the definition in (86). The time stamp can be set to the current time, and retirement-rate estimates can be set to current values.

Example 1. Consider a system with four different modes of operation. Let the corresponding performance-model coefficients be 1×10^6, 2×10^6, 4×10^6, and 8×10^6, respectively. Let the time spent in each mode be 0.3, 0.2, 0.2, and 0.3, respectively. These times and the performance-model coefficients are scaled by the length of the reservation period, T_R. Let t equal zero at the start of the reservation period. As

mentioned above, the implementation of a VIC source based on (8) requires maintaining some state. The successive updates to that state over the reservation period are shown in Table 8.

Table 8: State updates for a VIC source over a reservation period for Example 1.

t	t_0	Next t_0	$\hat{\alpha}_{m(t_0)}[k]$	Next $\hat{\alpha}_{m(t_0)}[k]$	$VIC(t_0)$	Next $VIC(t_0)$
0	-	0	-	1×10^6	-	0
0.3	0	0.3	1×10^6	2×10^6	0	$0 + 0.3 \times (1 \times 10^6)$
0.5	0.3	0.5	2×10^6	4×10^6	3×10^5	$3 \times 10^5 + 0.2 \times (2 \times 10^6)$
0.7	0.5	0.7	4×10^6	8×10^6	7×10^5	$7 \times 10^5 + 0.2 \times (4 \times 10^6)$
1.0	0.7	1.0	8×10^6	1×10^6	1.5×10^6	$1.5 \times 10^6 + 0.3 \times (8 \times 10^6)$

The piecewise-linear increase in $VIC(t)$ over time is shown in Figure 35. The points where mode transitions occur are marked with dashed lines in the figure.

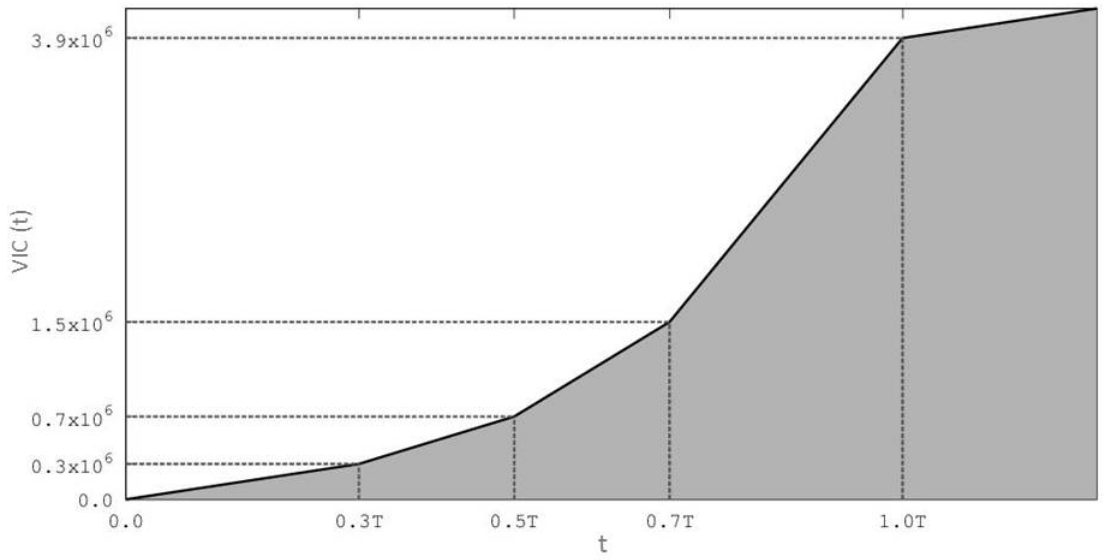


Figure 35: Plot of $VIC(t)$ against time over a single reservation period for Example 1. Mode transitions are marked with dashed lines.

5.2.3 VIC-based Budget

A VIC-based budget mechanism can be implemented in the same way as a time-based budget mechanism. A budget mechanism consists of two main components: a mechanism for measuring how much budget has been consumed by a task and a mechanism to prevent a task from using more than the allocated budget. The amount of budget consumed by a task can be measured as the amount by which the VIC source increases while the task runs. Accordingly, when some budget, $Q[k]$, is allocated to a task, the task should be put to sleep when the the VIC source increases by the amount $Q[k]$ during the execution of the task.

Example 2. Consider the system described in Example 1. Consider two tasks scheduled on such a system: *Task 1* and *Task 2*. Let the budget allocated to Task 1 and Task 2 be 1.1×10^6 and 2×10^6, respectively. Assume that the budget needed for Task 2 to complete all pending computation is 1.2×10^6. Also, assume that the budget allocated to Task 1 is insufficient for Task 1 to complete. Figure 36 shows a possible schedule for the two tasks over the reservation period.

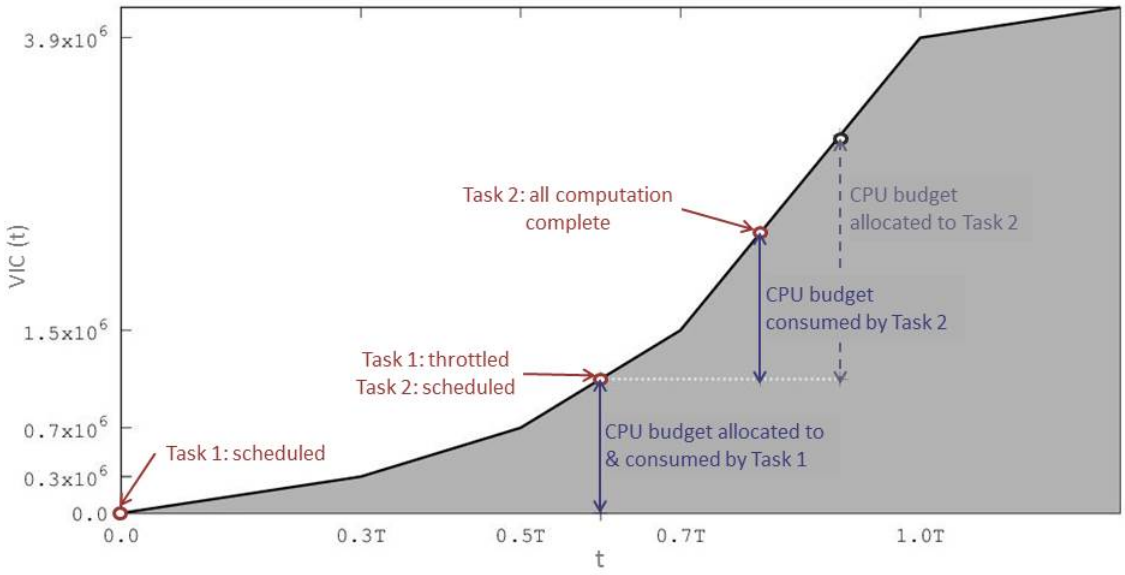


Figure 36: Plot of $VIC(t)$ against time over a single reservation period for Example 2. Scheduling events are marked with circles on the $VIC(t)$ line.

With VIC-based budget, the throttling time of tasks is based on $VIC(t)$. At time $t = 0$, Task 1 is scheduled to run, and $VIC(0) = 0$. At time $t = 0.6T$, $VIC(t)$ is 1.1×10^6, and Task 1 is still running. Since the budget for Task 1 is depleted, Task 1 is put to sleep and Task 2 is scheduled to run. At time $t = 0.8T$, $VIC(t)$ is 2.3×10^6. At this point, the budget consumed by Task 2 is $(VIC(0.8T) - VIC(0.6T)) = 1.2 \times 10^6$. Since enough budget is consumed that all pending computation can be completed, Task 2 is put to sleep until the following task period when a new job will be released. Had Task 2 continued to run, Task 2 would have been put to sleep at $t = 0.9T$ where $VIC(t) = 3.1 \times 10^6$.

5.2.4 Estimating Average Instruction-Retirement Rates

The next point of interest is how to update the estimates of the average instruction-retirement rate for each mode of operation. Let $\tau_1[k], ..., \tau_M[k]$ denote the amount of time the platform is configured to modes $1, 2, ..., M$, respectively, in the k^{th} reservation period. Let $\mu_m[k]$ denote the average instruction-retirement rate of the platform when configured to m^{th} mode. The number of instructions retired in the k^{th} reservation period, $I[k]$, can be expressed as follows:

$$I[k] = \sum_{m=0}^{M} \mu_m[k]\tau_m[k] \tag{87}$$

Equation 87 represents the performance model.

The performance-model coefficient, $\hat{\mu}_m[k]$, is an estimate of $\mu_m[k]$. As seen in Section 5.1.3, the instruction-retirement rate for the same workload in the same mode of operation can change with time. To adapt to the changing instruction-retirement rates, the normalized variable-step LMS algorithm is used as described in Section 4.2.2.

Specifically, $I[k]$ is measured using hardware-performance counters at the end of each reservation period. The time spent in each mode of operation during the previous reservation period is measured as well. The time spent in each mode of operation and

the performance-model coefficients are used to compute an estimated value of RIC, $\hat{I}[k]$:

$$\hat{I}[k] = \sum_{m=1}^{M} \hat{\mu}_m[k]\tau_m[k]. \tag{88}$$

This estimate is equal to the change in $VIC(t)$ over the duration of the reservation period; $\hat{I}[k] = VIC(T_R k) - VIC\left(T_R(k-1)\right)$. The error in the estimate, $e[k] = I[k] - \hat{I}[k]$, is used to adjust the performance-model coefficients as follows:

$$\hat{\mu}_m[k+1] = \hat{\mu}_m[k] + \beta_m[k] \left(\frac{-\tau_m[k]e[k]}{\sum_{i=1}^{M} \tau_i^2[k]} \right). \tag{89}$$

Finally, the step sizes, $\beta_m[k]$, are adjusted in the same way as described in Section 4.2.2, except that $\nabla_m \left(\mathbb{E}\left(e^2[k]\right)\right)$ is approximated as follows:

$$\nabla_m \left(\mathbb{E}\left(e^2[k]\right)\right) \approx -2e[k]\tau_m[k]. \tag{90}$$

An alternative approach for updating $\hat{\mu}_m[k]$ is to sample the performance counter at each mode transition and compute an exponentially-weighted moving average of the instruction-retirement rate for each mode of operation over time. However, depending on how often mode transitions take place, this approach can introduce a significant level of measurement overhead. In contrast, the approach described above based on the LMS-filter incurs a fixed amount of overhead in every reservation period.

5.2.5 Handling Overload Conditions

To ensure the feasibility of a set of budget allocations, the sum of the budget allocations across all hosted tasks must be less than an upper limit related to the utilization bound. This limit is called the *budget limit* in the remainder of this section. If the budget limit is violated, it is referred to as an overload condition as described at the end of Section 3.1.1. Overload conditions can be handled using the *compression*

approach, where all budget allocations are proportionally scaled back such that the total budget allocation is equal to the budget limit.

Budget allocations are adapted at the beginning of every reservation period. Therefore, overload conditions must be detected and handled at the beginning of every reservation period. The budget limit must be computed at the start of every reservation period. Obtaining the budget limit is trivial in the case of time-based budgets where the budget limit is equal to the length of the reservation-period, T_R. However, in the case of VIC-based budgets, the budget limit for the k^{th} reservation period is equal to $\hat{I}[k]$, the total increase in $VIC(t)$ over that reservation period. For example, for the reservation period from Example 1, the budget limit is 3.9×10^6.

The increase in VIC over a reservation period, $\hat{I}[k]$, is a function of the time spent in each mode of operation and the estimated instruction-retirement rate in each mode. This relationship is shown in (88). The retirement-rate estimates are computed using the LMS algorithm as described in Section 5.2.4. The amount of time to be spent in each mode of operation, $\tau_0[k]$, $\tau_1[k]$, ..., $\tau_M[k]$, constitutes the power-management decision.

If power-management decisions are made prior to budget-allocation decisions, then the values of $\tau_m[k]$ are fixed, and (88) can be used to compute the budget limit. On the other hand, if power-management decisions are made based on budget-allocation decisions, $\tau_m[k]$ values are flexible. In this case, the utilization bound should be set to $\hat{I}_{max}[k] = T_R\hat{\mu}_M[k]$, that is, the maximum possible value of $\hat{I}[k]$ that would result from spending the entire reservation period in the highest-performance mode.

5.3 Implementation of a VIC-Based Budget Mechanism

A VIC-based budget mechanism has been implemented for Linux by extending the TSP implementation described in Section 4.4. The overall structure is shown in Figure 37.

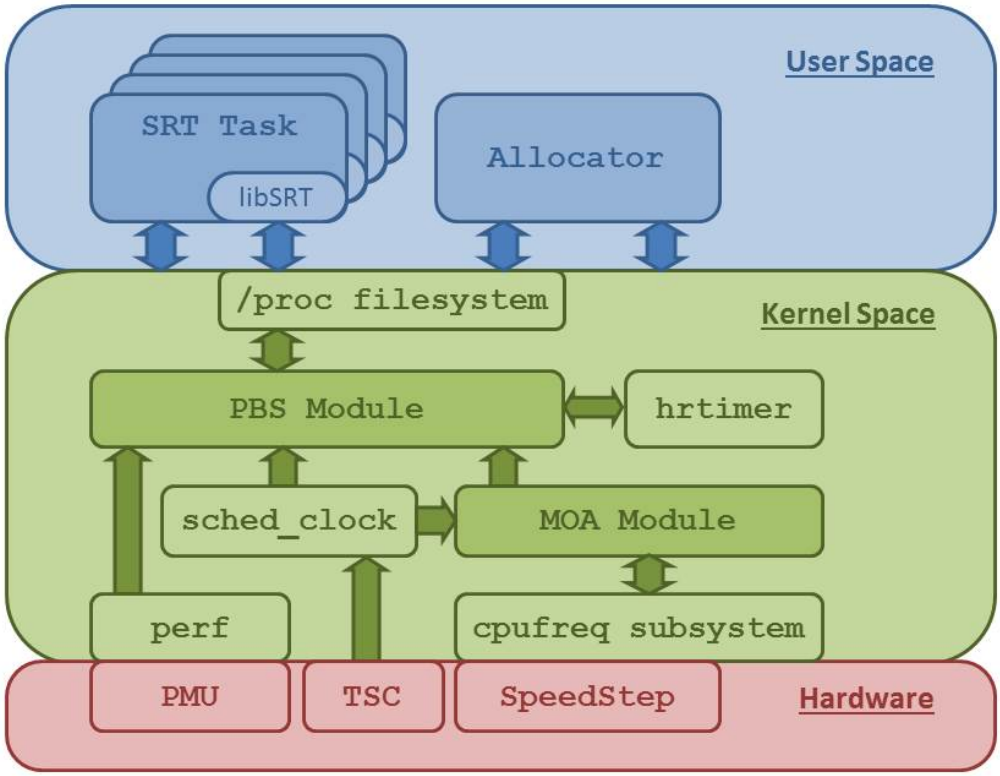


Figure 37: Overall structure of an implementation of the VIC-based budget mechanism in Linux.

5.3.1 SRT Tasks

VIC-based budget allocation requires very little change to SRT tasks. As described in Section 4.4, the first stage of the TSP algorithm is performed in the execution context of SRT tasks. In the first stage, the CPU usage of previous jobs are used to predict the CPU usage of future jobs and to estimate the statistics of the prediction error. With VIC-based budget allocation, this first stage remains the same except that CPU usage is measured and predicted in terms of VIC rather than CPU time.

5.3.2 The Allocator

The Allocator is a daemon process used to perform numerical operations in user space since floating-point operations are difficult to perform in the kernel. In the implementation of the TSP algorithm, the Allocator is used to perform the second

stage, computing the budget allocation for each task using current run times, job-queue lengths, and predicted execution times. For VIC-based budget allocation, the Allocator is used to perform additional computation.

Performance-model coefficients are updated in the Allocator at each reservation-period boundary. These coefficients represent the rate of increase in $VIC(t)$ over time and are used by the PBS module to implement the VIC source. At each reservation-period boundary, the PBS module provides the Allocator with the retired-instruction count over the previous reservation period and the amount of time spent in each mode of operation. Then, the LMS algorithm is used to update the performance-model coefficients as described in Section 5.2.4.

When performance-model coefficients are updated, the corresponding reciprocals are computed as well. The reciprocal of a performance-model coefficient is the amount of time needed for a unit increase in $VIC(t)$. These reciprocals are used in the PBS module to implement the VIC timer mechanism. The VIC timer is described in Section 5.3.4.

Once the budget allocation is computed for each task and the performance-model coefficients are updated, the Allocator must check for overload conditions as described in Section 5.2.5. If an overload condition is detected, the budget allocations are scaled back.

While computations in the Allocator are performed using floating-point operations, the output to kernel space can not remain in floating-point format. All budget allocations are converted to 64 bit integers. The performance-model coefficients and the corresponding reciprocals are converted to 64-bit fixed-point numbers with 48 fractional bits.

For efficient communication between the Allocator and the PBS module, all inputs and outputs of the Allocator are stored in shared memory pages that are mapped into both the Allocator address space and kernel-level address space.

5.3.3 The MOA Module

The modes-of-operation-abstraction (MOA) module abstracts the underlying power-management mechanism as discrete modes of operation. This generic abstraction layer provided by the MOA module allows for easier portability to other architectures and power-management mechanisms.

For the platform described in this section, the power-management mechanism is CPU-frequency scaling. In Linux, the CPU-clock frequency is managed through the `cpufreq` subsystem[40]. In the MOA module, the `cpufreq` subsystem is used to determine the number of CPU frequency settings available on the processor, to determine the active CPU frequency during initialization of the module, and for notifications when there are changes in the CPU frequency.

The MOA module assigns each frequency an integer index. The module keeps track of the index of the active frequency setting and the time spent in each frequency setting since initialization. This functionality is exported to the PBS module where it is used to implement the VIC-based budget mechanism. Furthermore, the MOA module exports a callback mechanism for mode-transition events. Whenever there is a change in the CPU frequency, the `cpufreq` subsystem notifies the MOA module. The MOA module calls any registered callback functions, providing the functions with the index of the preceding mode of operation and the following mode.

5.3.4 The PBS Module

The PBS module coordinates the timing of the Allocator daemon and all SRT tasks, managing the release of new jobs and tracking job-queue lengths. Also, enforcement of budget allocations and measurement of budget usage is managed through the PBS module. In this regard, the PBS module is the same as in the implementations described in Sections 3.5.2 and 4.4. However, to enable VIC-based budget allocation, the module is modified in a number of ways.

Allocator Feedback

The PBS module provides the Allocator with the data necessary to update the performance-model coefficients. This data includes the number of instructions retired over the previous reservation period and the amount of time spent in each mode of operation. The retired-instruction count is measured using the `perf` subsystem. The time spent in each mode of operation is obtained from the MOA module.

The VIC Source

The PBS module implements a *VIC source*, a monotonically increasing counter similar to a clock source. To implement the VIC source, the PBS module maintains some state as described in Section 5.2.2. This state is updated after two types of events: 1) when there is a change in the mode of operation and 2) at reservation-period boundaries after the performance-model coefficients are updated.

As mentioned in Section 5.3.3, the MOA module implements a callback mechanism for mode-transition events. This callback mechanism is used to update the state of the VIC source when there is a change in the CPU frequency. Also, the state of the VIC source is updated when the Allocator returns control to the PBS module after updating the performance-model coefficients.

VIC Timers

The PBS module implements a VIC callback mechanism called *VIC timers*. Similar to regular timers, a VIC timer can be armed to fire when the VIC source reaches a target value. This mechanism is built as a wrapper around the high-resolution-timer mechanism in Linux, `hrtimer`. When a VIC timer is armed, an hrtimer is armed based on the approximate time to target. The approximate time to target is computed as the product of the VIC to target and the reciprocal of the performance-model coefficient described in Section 5.3.2. The VIC to target is the difference between the target of the VIC timer and the current value of the VIC source.

When there is a change in the mode of operation, there is a change in the performance-model coefficient. As a result, there is a change in the actual time to target. The MOA module notifies the PBS module when there is a change in the mode of operation. Then, the PBS module iterates through the list of active VIC timers and recomputes the time to target for each VIC timer. The hrtimer associated with each of these VIC timers is reprogrammed and armed accordingly.

Measuring Budget Consumption & Enforcing Budget Allocations

All SRT tasks register with the PBS module during initialization. During registration, a `preempt_notifier` is attached to each task. A `preempt_notifier` is a callback mechanism for scheduling events; callback functions are called when the task associated with the `preempt_notifier` is scheduled to run or when that task is preempted. The `preempt_notifier`s allow the PBS module to sample the VIC source at the start and end of each task activation. The difference in the sampled values is the amount of CPU budget consumed by the task during that activation.

Budget allocations must be enforced. An SRT task must be put to sleep when the consumed CPU-budget reaches or exceeds the allocated budget. To enforce budget allocations, VIC timers are used. The PBS module to arms a VIC timer when an SRT task is scheduled to run. The target of the VIC timer is set to the current value of the VIC source plus the remaining budget. If the task completes all pending computation before the VIC timer fires, the VIC timer is disarmed. However, if the VIC timer fires and the budget remaining is found to be less than a threshold value, the task is put to sleep. Then, the task is awakened when the budget is refreshed in the following reservation period.

5.3.5 Handling an Idle CPU

The implementation of LAMbS described in this section does not handle sleep states. Specifically, if the processor enters a sleep state, the time spent in the sleep state is

attributed to the mode of operation that was active before the system entered the sleep state. The instruction-retirement rate in a sleep state is zero. As a result, the instruction-retirement rate of the affected mode of operation can be underestimated. In the extreme case, the performance-model coefficient of the affected mode is computed to be zero. This extreme case is especially problematic because the corresponding reciprocal of the coefficient is undefined, and the VIC timer and VIC budget mechanisms no longer function correctly.

The Linux kernel can be configured to prevent the system from entering a sleep state. Specifically, the kernel can be configured to use polling idle states using the boot time parameter "`idle=pol`". In this configuration, whenever the system is idle, the kernel enters a spin loop waiting for the arrival of an interrupt. The kernel never enters a hardware sleep state and the instruction-retirement rate is never zero.

However, the instruction count reported by the PBS module to the Allocator is user-level instruction count. As a result, instructions retired in the kernel-level spin loop is not counted towards the reported instruction count. Two possible approaches to address this problem are to handle sleep states as additional modes of operation and to maintain a separate set of performance-model coefficients for each SRT application. These two approaches are discussed in greater detail in Chapter 6. For the purposes of experiments described in Section 5.4, experiments were performed with a non-real-time task running in the background. This task served as an idle process. Because the task was scheduled as a non-real-time task, it did not interfere with the schedule of the Allocator or SRT tasks.

5.4 Experiments and Results

Results from a number of experiments demonstrate the advantages of VIC-based budget allocation over time-based budget allocation. Namely, the average per-job CPU usage is not affected by changes in the mode of operation when CPU usage is

measured in terms of VIC. As a result, when there are mode transitions, it is easier to predict CPU usage in terms of VIC rather than CPU time.

With an adaptive budget-allocation algorithm like TSP, the system eventually adapts to changes in job-execution times that results from changes in the mode of operation. However, with budget allocation based on VIC, the response to such changes is faster. When there are large and frequent changes in the mode of operation, VIC-based budget allocation is more efficient than time-based budget allocation.

All results presented in this section are based on experiments performed on the platform described in Section 5.1. Also, for reasons described in Section 5.3.5, all experiments were performed with an idle process running in the background.

5.4.1 Performance Metrics

Time-based budget and VIC-based budget cannot be compared directly. To compare the performance of these two approaches of budget allocation, three different metrics can be used: VFT error, deadline-miss rate, and average CPU-bandwidth allocation.

VFT error, also referred to as the scheduling error, is the difference between the virtual finishing time (VFT) of a job and the job deadline. Virtual finishing time of a job, $VFT[j]$, is the time a job would finish if it ran on a dedicated virtual processor whose speed is a fraction of that of the physical processor. The fraction is equal to CPU bandwidth. An expression for $VFT[j]$, first presented in Section 3.1.2, is repeated below for reference.

$$VFT[j] = LFT[j] - \frac{Q_{left}[j] \cdot T_R}{Q[k]}. \qquad \text{(9 revisited)}$$

The latest possible finishing time of a job, $LFT[j]$, is the end time of the reservation period in which the job completes. The variable $Q[k]$ is the budget allocated to the task in the reservation period in which the job completes. The variable $Q_{left}[j]$ is the amount of unused budget remaining when the job completes. Finally, T_R denotes the length of a reservation period. VFT error is often normalized by the task period.

VFT error is described in greater detail in Section 3.1.2.

The definition of VFT is based on the *ratio* of the allocated budget and the remaining budget. Therefore, VFT and VFT error are unaffected by the measure of computation used for budget allocation. VFT error is a performance metric that allows for comparison between time-based budget allocation and VIC-based budget allocation. The same is true for **deadline-miss rate** as a performance metric.

CPU bandwidth is the ratio between the allocated budget and the total CPU capacity available for allocation over the reservation period. For time-based budget allocation, total available CPU capacity is the length of the reservation period, T_R. Then, the bandwidth is defined as $Q[k]/T_R$. For VIC-based budget allocation, the total available CPU capacity is the increase in $VIC(t)$ from the beginning of the reservation period until the end of that reservation period, $(VIC((k+1)T_R) - VIC(kT_R))$.

Provided that the mode of operation remains constant within a reservation period, the CPU bandwidth is unaffected by the measure of computation used for budget allocation, CPU time or VIC. Like VFT error and deadline-miss rate, the average CPU-bandwidth allocation is a performance metric that allows for comparison between time-based budget allocation and VIC-based budget allocation.

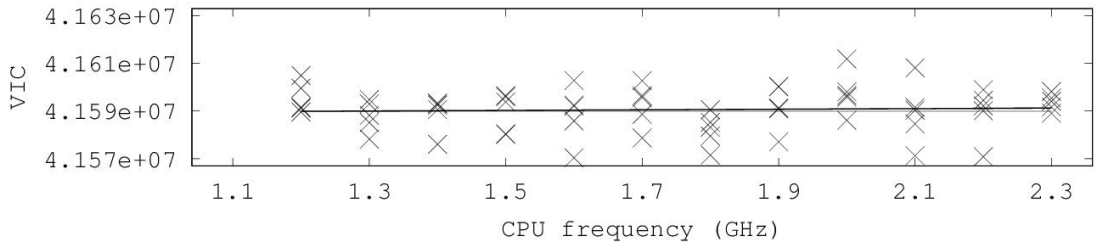
5.4.2 Invariance Under Change in The Mode of Operation

Per-job CPU usage, measured in terms of VIC, shows little variation with change in the mode of operation. The experiment described in Section 5.1.2 was repeated with VIC as the measure of computation. The average per-job CPU usage was measured for the three workloads. For each workload, the experiment was repeated at different CPU frequencies.

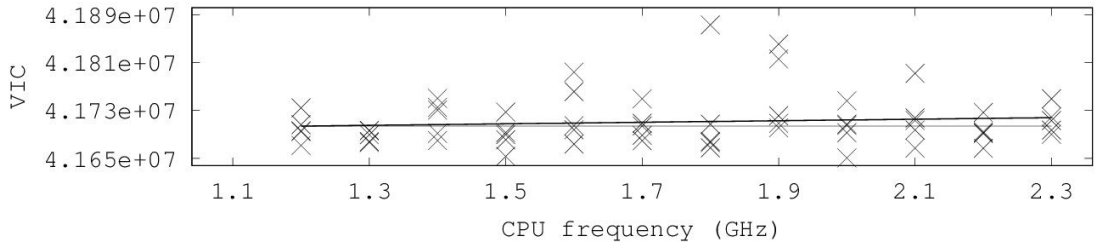
CPU usage was measured using the VIC source described in Section 5.3.4. The VIC source was sampled at the start and end of each job, and the difference in the two samples was used as the measure of CPU usage for that job. The average CPU

usage was computed across all jobs in the workload.

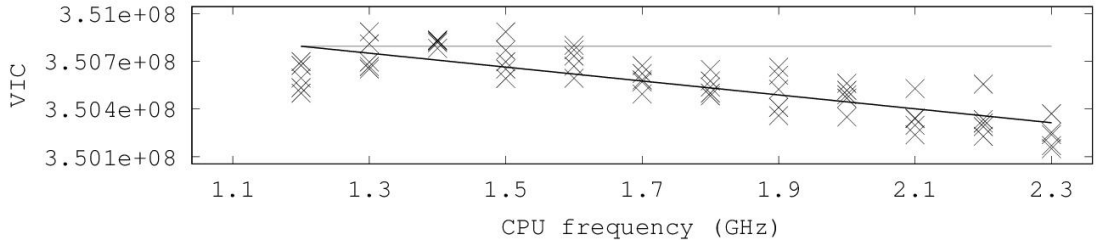
For each workload and CPU frequency, the experiment was repeated five times. The results are shown in the three plots in Figure 38. Each marker in these plots corresponds to the average CPU usage for a single run. The thick black lines are lines of best-fit. The thin gray lines correspond to ideal lines assuming perfectly-invariant CPU usage across all CPU frequencies.



(a) membound "cache line"



(b) membound "thrash"



(c) H.264 Video Decoding

Figure 38: Plots of average per-job CPU-usage (measured in VIC) against CPU-clock frequency for three different workloads.

For both membound workloads, there is negligible variability in the average per-job CPU usage. For the video-decoding workload, there is a decrease in CPU usage with increasing CPU frequency. As in the case of CPU cycles and retired-instruction count seen in Figures 30, 31, and 32, this decrease in the VIC can be attributed

to a decreasing number of interrupts. VIC is computed based on CPU time which includes kernel-level CPU time. As a result, VIC includes computation performed at the kernel level. However, this decrease in CPU usage with CPU frequency is negligible compared to overall CPU usage.

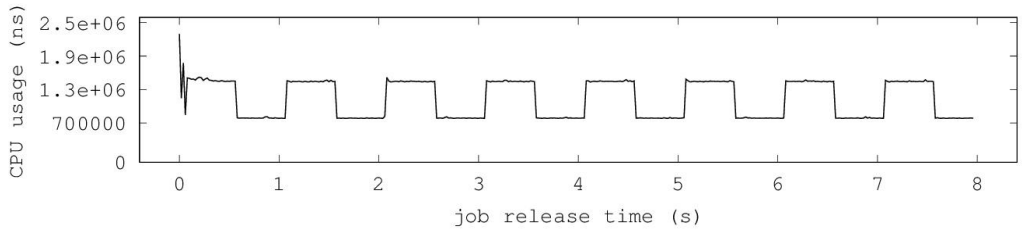
5.4.3 Response Time to Changes in The Mode of Operation

As shown in Section 5.1.2, per-job CPU usage changes with CPU frequency when CPU usage is measured in terms of execution time. Adaptive budget-allocation algorithms such as TSP can adapt to such changes in execution times. However, when CPU usage is measured in terms of VIC, there are no significant changes in per-job CPU usage with changes in CPU frequency. As a result, when budget allocations are based on VIC rather than execution time, the scheduling error due to such changes is significantly smaller.

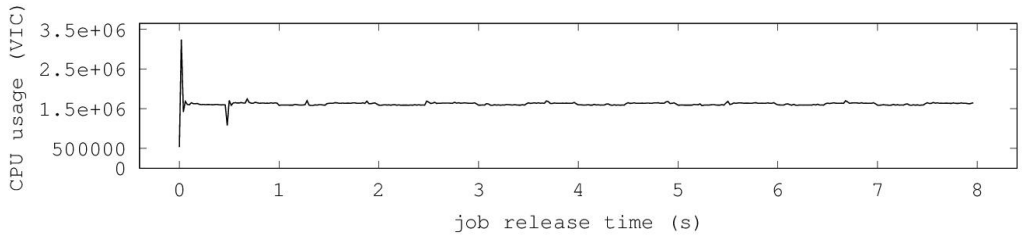
Experiments were performed using the membound "thrash" workload described in the Section 5.1.2. This synthetic workload consists of 400 jobs and each job consists of the same number of instructions. During the experiment, the CPU frequency was switched between $1.2GHz$ and $2.3GHz$ every half a second. The per-job CPU usage was measured in terms of both CPU time and VIC.

CPU usage is plotted against job-release times in Figure 39. As expected, there are step changes in the job-execution times due to changes in the CPU frequency. For CPU usage measured in terms of VIC, there are two initial spikes at $t = 0.0s$ and $t = 0.5s$. These two times represent the first time the workload runs at each of the two frequencies. The spikes correspond to an initial "bad guess" of the instruction-retirement rates in the two frequencies. Once the system "learns" the retirement rates there is very little change in the CPU usage across the remaining of jobs.

The membound "thrash" workload was scheduled with budget allocations based on the TSP algorithm. The length of the reservation period and task period was set



(a) job-execution time



(b) VIC

Figure 39: Per-job CPU usage for the membound "thrash" workload. The CPU frequency was switched between $1.2GHz$ and $2.3GHz$ every $0.5s$. CPU usage is shown in terms of job-execution times (top), and VIC (bottom).

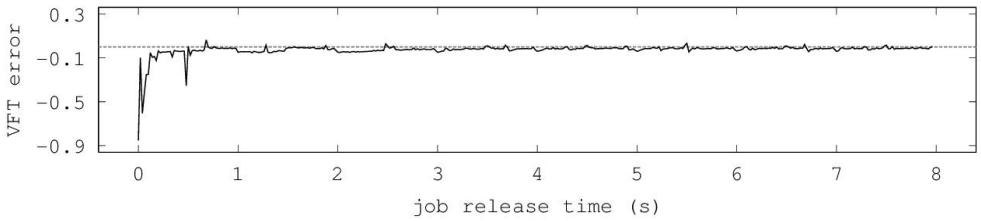
to $10ms$ and $20ms$, respectively. The alpha parameter was set to 1.25. The workload was tested with both time-based budget allocation and VIC-based budget allocation. The per-job scheduling error is plotted in Figure 40 against job release times.

With time-based budget allocation, there are positive and negative spikes in the scheduling error corresponding to positive and negative steps in job-execution times. After each spike, the scheduling error converges towards zero as budget allocations are adapted to changes in job-execution times. With VIC-based budget allocation, there are spikes in the VFT error at $t = 0.0s$ and $t = 0.5s$. These spikes correspond to the spikes in the per-job VIC shown in Figure 39b. After $t = 0.5s$, the VFT error remains close to zero unaffected by the changes in CPU frequency and execution time.

5.4.4 Bandwidth Allocation vs Deadline-Miss Rate

In the TSP algorithm, as the variance in job-execution times increases, budget allocations increase proportionally by an amount that depends on the alpha parameter.

(a) time-based budget



(b) VIC-based budget

Figure 40: VFT error for the workload shown in Figure 39 with time-based budget allocation (top) and VIC-based budget allocation (bottom).

Changes in job-execution times due to changes in CPU frequency increase the variance in job-execution times and as a result, increase the average budget allocation. Furthermore, this increase in average budget allocation may not yield a commensurate decrease in the deadline-miss rate. In contrast, with VIC-based budget allocation, there is little or no increase in the average budget allocation due to changes in the CPU frequency.

5.4.4.1 Experimental Setup

To explore the effects of frequency scaling on budget allocation, experiments were performed on the video-decoding workload described in Section 5.1.2. Experiments were performed with both time-based budget allocation and VIC-based budget allocation. The performance of the budget-allocation mechanism was measured in terms of average deadline-miss rate and average CPU-bandwidth allocation.

Budget allocation was based on the TSP algorithm. The frame rate of the decoded video was $24fps$. Therefore length of the reservation period and task period was set to $10.416667ms$ and $41.666668ms$, respectively. The alpha parameter from the TSP

algorithm was varied from 0.1 to 3.0. For higher values of the alpha parameter, budget allocation is higher and the average deadline-miss rate is lower. For a given deadline-miss rate, it is desirable to have lower budget allocations.

For each value of alpha, the experiment was performed under four different frequency-scaling configurations. In three of these configurations, the CPU frequency was set to a constant value of $1.2GHz$, $1.8GHz$, and $2.3GHz$, respectively. In the fourth configuration the frequency was switched between $1.2GHz$ and $2.3GHz$ every half a second. The average CPU frequency in this fourth configuration was $1.75GHz$.

5.4.4.2 Results

The effect of frequency scaling on CPU-bandwidth allocation and deadline-miss rate is shown in the scatter plots in Figures 41 and 42, respectively. Each marker in these plots corresponds to a single experiment defined by the alpha parameter, the type of budget allocation, and the frequency-scaling configuration. Different frequency-scaling configurations are shown in different colors. The 'X' markers correspond to experiments with VIC-based budget allocation. The 'O' markers correspond to experiments with time-based budget allocation. For each type of budget allocation and frequency-scaling configuration, larger values of alpha correspond to lower deadline-miss rates and higher average bandwidth allocations.

Results for the constant-frequency experiments are shown in Figure 41. At higher CPU frequencies, the average CPU-bandwidth allocation is lower for the same deadline-miss rate regardless of the type of budget allocation. For time-based budget allocation, the average bandwidth allocation is lower because there is a decrease in execution time while the length of the reservation period stays constant. For VIC-based budget allocation, the per-job VIC remains the same regardless of CPU frequency. However, since the performance-model coefficient is larger at higher frequencies, the total

available CPU capacity is larger at higher frequencies. As a result, the average CPU-bandwidth allocation is lower. Also, for any CPU frequency, if the frequency remains constant, the bandwidth-vs-miss-rate curve is roughly the same for both time-based budget allocation and VIC-based budget allocation.

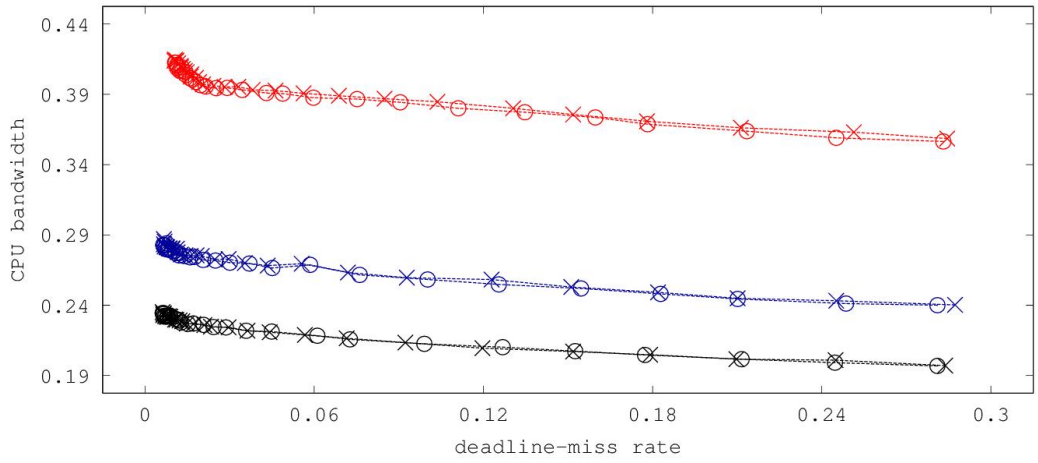


Figure 41: Scatter plot of average bandwidth allocation vs deadline-miss rate for a video-decoding workload with time-based budget allocation (O) and VIC-based budget allocation (X). CPU frequency was kept constant at $1.2GHz$ (red), $1.8GHz$ (blue), and $2.3GHz$ (black).

Results for the fourth frequency-scaling configuration are shown in Figure 42. Although the performance of time-based budget allocation and VIC-based budget allocation is similar when the CPU frequency is constant, VIC-based budget allocation performs better when there are changes in the CPU frequency. Specifically, for the same deadline-miss rate, the average bandwidth allocation is lower for VIC-based budget allocation than for time-based budget allocation.

5.5 Conclusion

For budget allocation in power-managed systems, virtual instruction count is a better measure of computation than CPU time, CPU cycles, or retired instruction count. CPU usage measured in terms of CPU cycles or CPU time changes with the mode of operation. As a result, with budget allocation based on CPU time or CPU cycles,

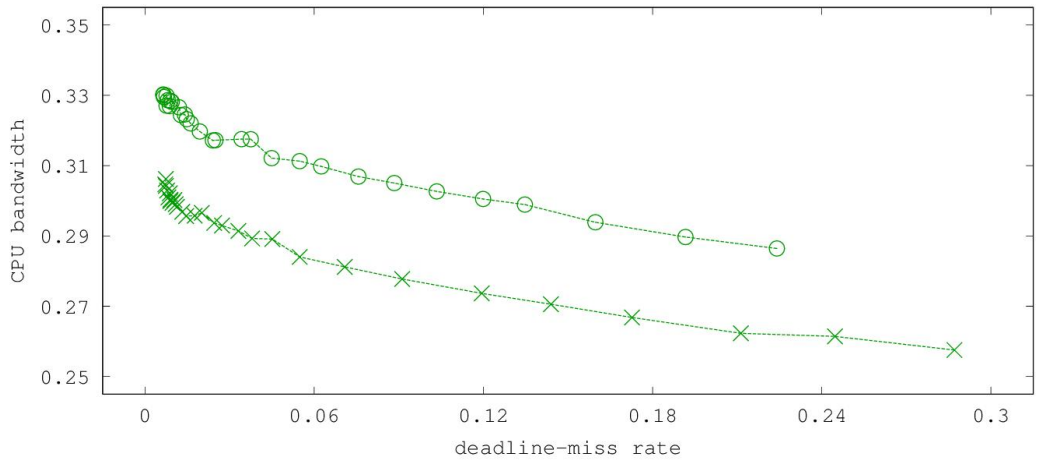


Figure 42: Scatter plot of average bandwidth allocation vs deadline-miss rate for a video-decoding workload with time-based budget allocation (O) and VIC-based budget allocation (X). CPU frequency was switched between $1.2GHz$ and $2.3GHz$ every half a second.

there are spikes in the VFT error with changes in the mode of operation.

With VIC-based budget allocation, such transients are smaller or no longer present. CPU usage measured in terms of VIC is unaffected by changes in the mode of operation. With no significant changes in CPU usage, the VFT error remains low. As a result, VIC-based budget allocation is more efficient, allowing for lower bandwidth allocations for the same deadline-miss rate.

CHAPTER VI

FUTURE WORK

In Chapters 3, 4, and 5, techniques are presented for dynamic allocation of CPU capacity to soft-real-time tasks, and mechanisms are described for enforcement of such allocations. In this chapter, power-management is discussed in greater detail and additional improvements are discussed for future work.

An adaptive power model is presented in Section 5.2.4 similar to the performance model described in Section 6.1. Based on this power model, the power-management problem is posed as a linear-programming problem.

The use of idle states for power management is discussed in Section 6.2. Experimental results are presented to demonstrate the potential benefits of power management with idle states. The power model from Section 6.1 is extended to include transition overheads. Based on the extended model, the power-management problem is reframed as a mixed-integer linear-programming problem. Implementation issues are also discussed.

The effects of nonuniform workload mixes are discussed in Section 6.3. When tasks hosted on the system have very different instruction-retirement rates, VIC is no longer a good approximation of instruction count. The invariance property of VIC discussed in Chapter 5 is compromised. As a result, VIC-based budget allocation does not perform well with workload mixes with a wide range of retirement rates. Experimental results are presented to demonstrate this effect, and potential solutions to the problem are discussed.

6.1 Power Management Using Linear Models and Constraints

In LAMbS, power-management decisions are in the form of *operation-mode schedules*. An operation-mode schedule is a vector, $\vec{\tau}[k] = \{\tau_1[k], \tau_2[k], ..., \tau_M[k]\}^T$, that specifies how much time should be spent in each mode of operation during a reservation period. The variable, $\tau_m[k]$, denotes the amount of time to be spent in the m^{th} mode of operation in the k^{th} reservation period.

Each SRT task is allocated some amount of CPU capacity to run over the duration of a reservation period. Taking into account the total CPU capacity committed across all tasks, any power-management decision must ensure that those commitments can be kept. An optimum operation-mode schedule is one that allows all budget commitments to be kept while minimizing the estimated energy usage for the corresponding reservation period.

6.1.1 The Power Model

To estimate the energy usage over a reservation period, a linear model is used similar to the performance model. The average power-consumption rate for each mode of operation is dynamically estimated. These estimates are referred to as power-model coefficients. The power-model coefficient for the m^{th} mode of operation during the k^{th} reservation period is denoted $\hat{p}_m[k]$. Energy usage over a reservation period is estimated as follows:

$$\hat{E}[k] = \sum_{m=1}^{M} \hat{p}_m[k]\tau_m[k]. \tag{91}$$

Let $p_m[k]$ denote the actual average power-consumption rate for the m^{th} mode of operation in the k^{th} reservation period. Then, the actual energy usage over the k^{th} reservation period is expressed as follows:

$$E[k] = \sum_{m=1}^{M} p_m[k]\tau_m[k]. \tag{92}$$

At the end of each reservation period, the energy usage is measured. Based on this measurement, the error is computed for the estimated energy usage, $e[k] = E[k] - \hat{E}[k]$. The power-model coefficients are adjusted based on this error value using the LMS algorithm as shown below.

$$\hat{p}_m[k+1] = \hat{p}_m[k] + \beta_m[k] \left(\frac{-\tau_m[k]e[k]}{\sum_{i=1}^{M} \tau_i^2[k]} \right). \tag{93}$$

Finally, the step sizes, $\beta_m[k]$, are adjusted in the same way as described in Sections 5.2.4.

Implementation

The LMS filter is already implemented in the Allocator for updating the performance-model coefficients. The same filter implementation can be used to update power-model coefficients using a separate filter state. However, to update the power-model coefficients, energy usage must be measured over each reservation period.

Instruction counts are measured using hardware performance counters. Such performance counters are a part of the processor and can be sampled relatively frequently with little overhead or delay. On the other hand, power and energy measurements are often performed further from the processor. In previous work, power or energy measuremets have been performed using external hardware such as multimeters, oscilloscopes, and power supplies [74][73][69]. Such approaches involve large delays and require large sampling periods.

On Intel processors based on the Sandy Bridge architecture or newer architectures, energy-usage measurements are made internally and can be accessed every millisecond through the Running Average Power Level (RAPL) interface [41]. These measurements are read through a model-specific register (MSR). The PBS module can obtain the energy measurements by reading the relevant MSRs and can pass

these measurements to the Allocator through the shared memory pages described in Section 5.3.2.

With a fast enough sampling rate, it may be possible to estimate the power-model coefficients using exponentially-weighted moving averages. However, the LMS filter approach described above allows for the coefficients to be computed with a single measurement and a fixed amount of computation every reservation period. As a result, the LMS filter approach has a lower overhead.

6.1.2 The Optimum Operation-Mode Schedule

Given the power model, performance model, and total budget allocation across all SRT tasks, the optimum operation-mode schedule for the k^{th} reservation period is one that minimizes the estimated energy usage over that reservation period. The cost function in this case is defined by Equation 91.

The length of the reservation period is fixed at T_R. As a result, the following constraint is imposed on the operation-mode schedule:

$$\sum_{m=1}^{M} \tau_m[k] = T_R. \tag{94}$$

Also, it must be possible for all SRT tasks to consume the allocated budget within the reservation period. Therefore, the increase in $VIC(t)$ from the beginning of the reservation period until the end of the reservation period must be greater than the total VIC budget allocated across all SRT tasks. Let $Q_{total}[k]$ denote the sum of the budget allocation across all SRT tasks after any overload conditions have been handled. Based on Equation 88, this minimum-performance constraint can be expressed as follows:

$$\sum_{m=1}^{M} \hat{\mu}_m[k]\tau_m[k] \geq Q_{total}[k]. \tag{95}$$

Given the linear cost function and linear constraints, this optimization problem can be solved using a standard linear-programming algorithm. Provided that overload

conditions are handled appropriately, $Q_{total}[k]$ is always less than or equal to $T_R\hat{\mu}_M[k]$. As a result, there is always at least one solution to the constraints, (94) and (95).

Implementation

The operation-mode schedule must be computed at the start of every reservation period. Furthermore, the optimization problem described above requires access to the power-model coefficients, performance model coefficients, the total budget allocation across all tasks, and floating-point operations. Therefore, the optimization should be performed in the execution context of the Allocator daemon. One possible library that can be used to solve the linear-programming problem is the GNU Linear-Programming Kit (GLPK) [1].

Since the operation-mode schedule must be computed at the start of every reservation period, the computational overhead is relevant. On most processors the number of modes of operation is relatively small. For the platform described in Table 6, the processor is configurable to 13 different frequency settings. Also, the optimization problem described in this Section 6.1.2 consists of just two constraints in addition to the cost function. Given the small size of the problem, the computational overhead should be acceptable.

Once computed, the operation-mode schedule can be passed to the kernel through the shared memory pages described in Section 5.3.2. Then, a kernel level component, referred to as the *operation-mode scheduler*, can configure the processor to the different modes of operation for appropriate durations of time based on the operation-mode schedule. In Linux, the CPU-clock frequency is managed though the `cpu-freq` subsystem [40]. The frequency is set by components called `cpu-freq` *governors*. The operation-mode scheduler can be implemented as such a governor.

6.2 Power Management with Idle States

The power-management problem discussed in Section 6.1 is limited to active modes of operation such as CPU-frequency settings. However, restricting power-management decisions to active modes of operation alone can limit energy savings.

To measure the relationship between CPU frequency and CPU power consumption, experiments were performed with a video-decoding workload and the platform described in Table 6. The workload was setup to decode one frame after the other without blocking and run repeatedly over a duration of one minute. The CPU frequency was kept constant over that interval.

Energy measurements were taken using the RAPL interface at the start and end of the one-minute interval. The sample times were recorded as well to measure the exact length of the intervals. Measurements of energy consumption and interval-lengths were used to compute the average power-consumption rate. For each CPU frequency, the experiment was repeated five times and the average power-consumption rate was measured for each repetition. The measurements are shown in Figure 43. A line of best fit is drawn through the markers and extrapolated to the y-axis.

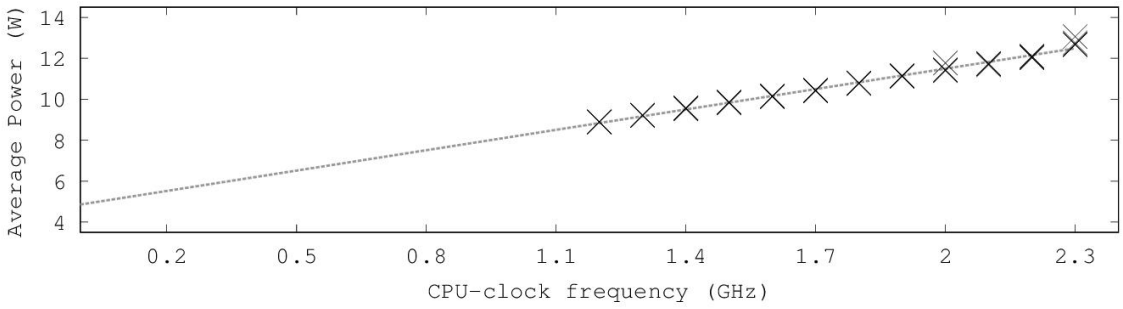


Figure 43: Average power-consumption rate at different CPU clock frequencies for a video-decoding workload.

Based on the figure, the relationship between frequency and power-consumption rate is linear and not quadratic. This suggests that the underlying power-management mechanism in the experimental platform is based on frequency scaling alone and not

based on voltage scaling. Furthermore, based on extrapolation of the data, the use of idle states can yield significant energy savings beyond what is possible with frequency-scaling alone.

However, idle states have transition overheads. Exiting from an idle state often requires additional time and energy. The model presented in Section 6.1.1 does not take such overheads into account. In this section, the power-management problem is modified to take such overheads into account. Also, additional implementation-related issues are discussed for power management with idle states.

6.2.1 Addressing Transition Overheads

The overhead associated with idle-state transitions is in the form of transition latencies and fixed energy costs. The amount of time spent in an idle state must be zero or greater than the transition time. When the transition time is strictly positive, the *search space* of valid operation-mode schedules *is no longer connected*. Furthermore, the fixed energy cost of an idle-state transition is only incurred when the idle state is used. Therefore, energy is no longer a continuous function of the operation-mode schedule.

In previous sections, active modes of operation are indexed 1 through M where M corresponds to the highest-performance state. Maintaining this notation, idle modes of operation are indexed $(-S+1)$ through 0 where S denotes the number of idle states and $(-S+1)$ is the index of the deepest idle state. For all idle states, the instruction-retirement rate is zero. Deeper idle states have higher transition overheads.

The variable, $\tau_m[k]$ denotes the time spent in the m^{th} mode of operation. In addition, let $o_m[k]$ denote an indicator variable for the m^{th} mode of operation. This variable is equal to 1 if the platform is configured to the m^{th} mode of operation in

the k^{th} reservation period and 0 if this mode of operation is not used.

$$o_m[k] = \begin{cases} 1 & \tau_m[k] > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{96}$$

Indicator variables are only needed for modes of operation with significant transition overheads. Let $\hat{E}_{\text{overhead},m}[k]$ denote the estimated energy cost of a transition to the m^{th} mode of operation. Equation 92, corresponding to estimated energy usage, is modified as follows to take the energy cost of a transition into account:

$$\hat{E}[k] = \sum_{m=-S+1}^{M} \hat{p}_m[k]\tau_m[k] + \sum_{m=-S+1}^{0} o_m[k]\hat{E}_{\text{overhead},m}[k]. \tag{97}$$

Due to the indicator variable, $o_m[k]$, the fixed energy overhead of the m^{th} mode of operation is only added to the estimated energy total if the mode of operation is used, $o_m[k] = 1$.

Let $t_{latency,m}$ denote the transition latency of mode m. When a mode of operation is used, the time spent in that mode must be greater than or equal to $t_{latency,m}$. Furthermore, when $o_m[k] = 0$, it must be ensured that $\tau_m[k] = 0$. These constraints are expressed as follows:

$$o_m[k]T_R \geq \tau_m[k] \geq o_m[k]t_{latency,m} \tag{98}$$

where T_R denotes the reservation-period length.

The length of the reservation periods, T_R, are known. The performance-model coefficients, $\mu_m[k]$, for the sleep states are zero, and the coefficients for the active modes of operation are adapted using LMS filters as described in Section 6.1.1. However, the three remaining sets of parameters, $\hat{p}_m[k]$, $\hat{E}_{\text{overhead},m}[k]$, and $t_{latency,m}$, may be more difficult to determine.

The power-model coefficients, $\hat{p}_m[k]$, and the estimated energy overhead of sleep-state transitions, $\hat{E}_{\text{overhead},m}[k]$, may be adapted using the normalized LMS algorithm.

However, values of $\tau_m[k]$ are specified in nanoseconds and tend to be large in comparison to $o_m[k]$, which can only be zero or one. As a result, the normalized LMS algorithm may be slow to converge to accurate estimates of $\hat{E}_{\text{overhead},m}[k]$ due to the normalization. In Linux, an alternative approach to obtain parameters such as $\hat{p}_m[k]$ and $t_{latency,m}$ is from `cpuidle` drivers. The `cpuidle` subsystem is described in greater detail in the following section.

6.2.2 Practical Considerations

For LAMbS to perform power-management using idle states, three main components are necessary. First, it must be possible to control the amount of time spent in a specific idle state within a reservation period. Second, it must be possible to perform the optimization described in Section 6.2.1 in every reservation period without significantly impacting the overall performance of the system. Lastly, it must be possible to obtain accurate estimates of the parameters needed to perform the optimization.

In Linux, idle states are managed through the `cpuidle` subsystem [58][25]. This subsystem consists of two main parts: `cpuidle` governors and `cpuidle` drivers. Governors implement the policy side of `cpuidle`[25]. Whenever there are no tasks to run, the system enters an idle period. The system will remain in the idle period until the arrival of an interrupt. A `cpuidle` governor is responsible for picking the idle state the system should enter at the start of an idle period. This decision can be made based on a number of heuristics, such as the time remaining until the next timer tick or the average length of previous idle periods. While the `cpuidle` governor controls which idle state is used, the governor has no control over when an idle period occurs or for how long the idle period lasts. A `cpuidle` driver is responsible for abstracting the idle states of the underlying hardware. Specifically, a `cpuidle` driver determines the number of idle states available in the hardware, and for each idle state, the driver maintains a data structure called `cpuidle_state` with relevant information on that

state.

As described in Section 6.1.2, power-management decisions in LAMbS are implemented using a kernel-level component called the operation-mode scheduler. To integrate idle states into LAMbS, it must be possible for the operation-mode scheduler to configure the system to an idle state when desired and to control how much time is spent in that state. The `cpuidle_state` data structure contains a pointer to a function that will switch the system into the corresponding idle state. This function should allow the system to enter an idle state regardless of the number of pending tasks. However, it is not clear how the system can be kept in the idle state for the desired amount of time.

Secondly, the optimization problem described in Section 6.2.1 is more compute intensive than the original problem described in Section 6.1.2. For each sleep state, two additional variables are introduced into the problem: $\tau_m[k]$ and $o_m[k]$. Furthermore, $o_m[k]$ is an integer variable. Also, in addition to the two constraints listed in Section 6.1.2, two more constraints are added to the problem for each sleep state. These additional constraints are defined by (98).

However, most processors have a small number of idle states. For the platform described in Table 6, the processor is configurable to four different C-states. Given the small number of idle states, it may still be computationally feasible to solve the MILP problem in every reservation period. The GLPK package mentioned in Section 6.1.2 is still applicable to the problem.

Another obstacle to solving the MILP problem is the need for parameters such as transition latency. One possible way to obtain these parameters is through the `cpuidle_state` data structure mentioned above. This data structure includes information such as the transition latency, power usage, and target residency of the associated idle state. Target residency refers to the break-even time for the corresponding state. These numbers are hard coded into architecture-specific parts of the

kernel source, such as the driver file `intel_idle.c`. However, this information is not always valid. When these parameters were inspected in the test platform using the `sysfs` interface in Linux ([58]), the reported power usage was incorrect.

To include idle states in power-management decisions in LAMbS, the issues discussed above must be resolved.

6.3 Workload Mixes with Nonuniform Retirement Rates

The improved performance of VIC-based budget allocation is due to invariance under change in the mode of operation. Specifically, VIC approximates RIC and per-job CPU-usage measured in terms of RIC is invariant under change in the mode of operation.

The approximation of RIC is based on the integration of the estimated average instruction-retirement rate over time. Inherent in this approximation is the assumption that for a given mode of operation, the instruction-retirement rate remains uniform throughout the duration of a reservation period.

However, as seen in Figure 33 in Section 5.1.3, instruction-retirement rates can differ greatly across workloads. When a system is shared between mixed workloads with a wide range of instruction-retirement rates, assumptions about uniformity in time are no longer valid. While the average instruction-retirement rate may be estimated correctly, the actual instruction-retirement rate may deviate significantly from the average. With an inaccurate estimate of the instruction-retirement rate, VIC does not track RIC well. In such cases, CPU-usage measured in terms of VIC is no longer invariant under change in the mode of operation.

To demonstrate this effect of mixed-retirement-rate workloads, experiments were performed using the system described in Section 5.3. The experiment was performed for two different workload mixes: one with a uniform instruction-retirement rate and one with a non-uniform retirement rate.

Each of these workload mixes consists of two SRT tasks. In both cases, the first SRT task consists of the "cacheline" configuration of the membound workload. In the uniform case, the second task also consists of the "cacheline" configuration of the membound workload. In the non-uniform case, the second task consists of the "thrash" configuration of the membound workload. As described in Section 5.1.3, the "cacheline" configuration of the workload is chip bound and has a significantly higher retirement rate than the "thrash" configuration which is memory bound. The two workload mixes are summarized in Table 9.

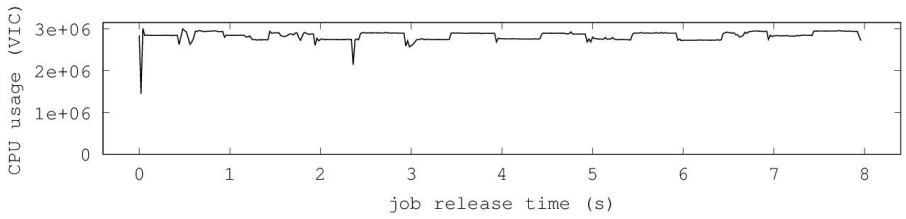
Table 9: Workload Mixes with Uniform and Nonuniform Instruction-Retirement Rates.

Workload Mix	SRT Task 1	SRT Task 2
Uniform	membound "cacheline"	membound "cacheline"
Nonuniform	membound "cacheline"	membound "thrash"

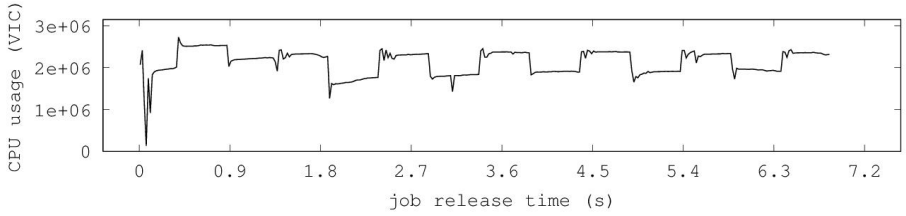
During the experiment, the CPU frequency was switched between $1.2GHz$ and $2.3GHz$ every half a second. The CPU usage of each job was measured in terms of VIC. In Figure 44, the CPU usage is plotted against release times for SRT Task 1 from both of the workload mixes. With a uniform instruction-retirement rate across the two tasks, there is little variability in VIC-based CPU usage despite changes in the CPU frequency. In contrast, with non-uniform instruction-retirement rates, there is significant change in CPU usage.

Unanticipated changes in CPU usage result in spikes in the VFT error. The VFT error for SRT Task 1 from the two workload mixes are shown in Figure 45. The VFT error from the nonuniform workload mix shows significant spikes corresponding to changes in the CPU frequency. In contrast, the VFT error from the uniform workload mix remains close to zero most of the time.

One way to address workload mixes with a wide range of instruction-retirement rates is to maintain a separate set of performance-model coefficients for each SRT



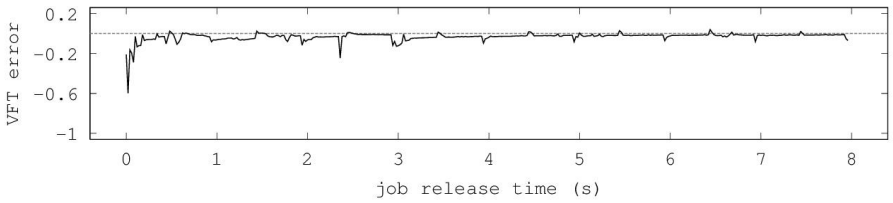
(a) Uniform



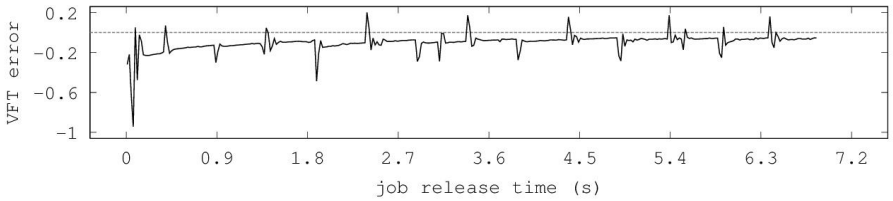
(b) Nonuniform

Figure 44: Per-job CPU usage vs job-release time for SRT Task 1 from the two workload mixes described in Table 9.

task. However, this approach increases the number of variables in the optimization problem by a factor equal to the number of tasks in the system. Another approach is to cluster together tasks with similar retirement rates and only maintain a separate set performance-model coefficients for each cluster. However, this approach would involve significant overheads as well.



(a) Uniform



(b) Nonuniform

Figure 45: VFT error vs job-release time for SRT Task 1 from the two workload mixes described in Table 9.

CHAPTER VII

SUMMARY

In this dissertation, a novel approach to power management is presented for soft-real-time applications, the Linear Adaptive Models-based System (LAMbS). This approach is based on dynamically adapting to the computational requirements of hosted applications and adapting to the power and performance characteristics of the underlying hardware.

LAMbS is designed for periodic real-time applications that consist of a timed sequence of jobs. The release times of jobs are known, but job-execution times are not known. Power management of such applications entails predicting the computational load of future jobs and configuring the underlying hardware such that power consumption is reduced and the predicted load is accommodated. The focus of this Dissertation is on predicting computational load.

Asynchronous Budget Allocation

In LAMbS, SRT tasks are scheduled with reservation-based schedulers. For such schedulers, time is divided into fixed intervals called reservation periods. Each task is allocated a budget to run within the reservation period. The allocated budget is dynamically adapted based on predicted computational requirements of current and future jobs.

One of the contributions of this dissertation is asynchronous adaptation of CPU budgets. Specifically, an algorithm is presented for adapting budget allocations at reservation-period boundaries. Previous algorithms for adaptive budget allocation are based on adaptations on job completion or adaptation on multiple job completions. The advantage of adapting at reservation-period boundaries is that it is possible to

respond much faster to significant increases in CPU usage. Simulation results are presented to demonstrate this advantage.

A proof is presented that asynchronous budget allocation maintains a stable job queue. A software architecture is presented for a Linux-based implementation of asynchronous budget allocation. Finally, experimental results are presented to demonstrate that this algorithm allows for rapid recovery from overload conditions. Also, results are presented to show that this algorithm performs well with video-decoding workloads and performs better than static budget allocation.

Two-Stage Prediction

Another contribution of this dissertation is the two-stage prediction (TSP) algorithm. In this algorithm, the problem of workload prediction is separated into two stages. An initial application-specific stage predicts the CPU usage of future jobs and estimates the mean and variance of the corresponding prediction error. The second stage uses the output of the first stage and additional feedback from kernel level components to compute the budget allocation. The first stage is synchrounous; the output is updated on job completion. The second stage is asynchronous and updates the budget allocation at reservation-period boundaries. Furthermore, the second-stage prediction algorithm is based on the Chebyshev inequality and is designed to bound the probability of a deadline miss.

A generic first-stage prediction algorithm is presented called the LMS-MA Hybrid predictor. The predictor consists of an array of normalized variable-step-size LMS filters of diferent lengths and an array of moving averages of different lengths. Based on measurements of execution times from a wide range of multimedia workloads, job-execution times are correlated for some workloads and uncorrelated for others. LMS filters are good predictors when execution times are correlated. Moving averages

perform better than LMS filters when execution times are uncorrelated. The LMS-MA Hybrid predictor was designed to perform well in both cases.

Details are presented for an implementation of the Two-Stage predictor in a Linux-based system. Results are presented from experiments with multimedia workloads. Generally, the performance of the two-stage predictor depends on the accuracy of the first-stage predictor. Also, the LMS-MA hybrid algorithm generally performs better with workloads where execution times are correlated.

Virtual Instruction Count

The third major contribution of this Dissertation is virtual instruction count (VIC). VIC is an abstract measure of computation based on approximating retired-instruction count (RIC). The number of retired instructions required to complete a job is invariant under change in the CPU frequency. However, the amount of time required to retire a speific number of instructions is hard to predict, which makes RIC unreliable as a measure of computation for budget allocation. In constrast, VIC increases over time at a known rate. Since VIC approximates RIC, VIC is invariant under change in CPU frequency to a degree that depends on the accuracy of the approximation. Therefore, VIC is a good measure of computation for budget allocation.

Implementation details are prsented for adaptive budget allocation based on VIC. Experimental results are presented with synthetic and video-decoding workloads. Performance of VIC-based budget allocation is compared to that of time-based budget allocation in terms of average bandwidth allocation, RMS VFT error, and deadline miss rate. VIC-based budget allocation performs similar to time-based budget allocation when there are no changes in CPU frequency. VIC-based budget allocation performs better when there are frequenct changes in CPU frequency.

The performance of VIC-based budget allocation in the presence of change in CPU frequency depends on the invariance property. The invariance property of VIC

depends on how well VIC approximates RIC. VIC approximates RIC using the average instruction-retirement rate. When the hosted set of tasks have a large range of retirement rates, the actual retirement rate may deviate significantly from the average. In such cases, VIC does not approximate RIC well and the invariance property is compromized. Experimental results are presented to demonstrate this effect. Possible solutions are proposed to address this problem in future work.

Power Management

Lastly, the power-management aspect of LAMbS is discussed for future work. In previous work, power-management decisions involve choosing the mode of operation to which the CPU should be configured at various stages of progress in the workload. In LAMbS, the power management decision involves computing the amount of time to spend in each mode of operation over periodic intervals. This decision is in the form of a vector called the operation-mode schedule.

A model is presented for power dissipation as a function of the operation-mode schedule. An algorithm is presented for correcting the coefficients of the power model based on run-time measurements of the system's power-usage. It is shown that computing the optimum operation-mode schedule that minimizes the predicted power dissipation is a linear-programming problem.

The problem is modified to allow for power-management decisions with idle states. The energy model is modified to account for transition overheads that are associated with idle states. Additional constraints are imposed on the operation-mode schedule to account for minimum transition latency. The power-management problem is reframed as an MILP problem. A Linux-based implementation of this power-management algorithm is discussed for future work.

Conclusion

LAMbS is a novel approach to power management. Due to the generic hardware model based on discrete modes of operation, LAMbS is applicable to a range of power-management mechanisms that go beyond frequency scaling. The adaptive components of LAMbS such as budget allocation, power model, and performance model, allows LAMbS to be used with no apriori knowledge and little or no tuning. Although further work is needed for a complete implementation, *LAMbS holds promise as a practical and portable system for real-time power management.*

REFERENCES

[1] "Glpk (gnu linear programming kit)." http://www.gnu.org/software/glpk/.

[2] *Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age White Paper*. Qualcomm, October 2011.

[3] *Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance (White Paper)*. NVIDIA, 2011.

[4] Abeni, L. and Buttazzo, G., "Integrating multimedia applications in hard real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, (Washington, DC, USA), pp. 4–, IEEE Computer Society, 1998.

[5] Abeni, L., Cucinotta, T., Lipari, G., Marzario, L., and Palopoli, L., "Qos management through adaptive reservations," *Real-Time Systems*, vol. 29, pp. 131–155, 2005.

[6] Abeni, L., Palopoli, L., Lipari, G., and Walpole, J., "Analysis of a reservation-based feedback scheduler," in *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pp. 71–80, 2002.

[7] Ahmed, S. and Ferri, B., "Prediction-based asynchronous cpu-budget allocation for soft-real-time applications," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, p. 1, 2013.

[8] Ahmed, S. and Ferri, B., "Prediction based bandwidth reservation," in *Decision and Control (CDC), 2010 49th IEEE Conference on*, pp. 5302 –5307, Dec. 2010.

[9] Ahmed, S. and Ferri, B., "Two-stage prediction for cpu time allocation in soft real-time systems," in *Feedback Computing, 2012 7th International Workshop on*, Sept. 2012.

[10] AMD, *AMD64 Architecture Programmers Manual Volume 2: System Programming*, revision 3.22 ed., Sept. 2012.

[11] Amirijoo, M., Hansson, J., Gunnarsson, S., and Son, S., "Enhancing feedback control scheduling performance by on-line quantification and suppression of measurement disturbance," in *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pp. 2–11, march 2005.

[12] Amur, H., Schwan, K., and Prvulovic, M., "Towards optimal power management: Estimation of performance degradation due to dvfs on modern processors," cercs technical report, Georgia Institute of Technology, 2010.

[13] Asberg, M., Nolte, T., Otero Perez, C., and Kato, S., "Execution time monitoring in linux," in *Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pp. 1–4, sept. 2009.

[14] Aydin, H., Melhem, R., Mosse, D., and Mejia-Alvarez, P., "Power-aware scheduling for periodic real-time tasks," *Computers, IEEE Transactions on*, vol. 53, pp. 584 – 600, May. 2004.

[15] Aydin, H., Devadas, V., and Zhu, D., "System-level energy management for periodic real-time tasks," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pp. 313 –322, Dec. 2006.

[16] Barroso, L. A. and Hölzle, U., "The case for energy-proportional computing," *Computer*, vol. 40, pp. 33–37, Dec. 2007.

[17] Bavier, A. C., Montz, A. B., and Peterson, L. L., "Predicting mpeg execution times," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, pp. 131–140, June 1998.

[18] Bellard, F., Niedermayer, M., and others, "Ffmpeg," *Internet: http://www. ffmpeg. org,[Dec. 27, 2012]*, 2007.

[19] Benini, L., Bogliolo, A., and De Micheli, G., "A survey of design techniques for system-level dynamic power management," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, pp. 299 –316, June 2000.

[20] Bilcu, R. C., Kuosmanen, P., and Egiazarian, K., "A new variable length lms algorithm: theoretical analysis and implementations," in *Electronics, Circuits and Systems, 2002. 9th International Conference on*, vol. 3, pp. 1031–1034 vol.3, 2002.

[21] Bini, E., Buttazzo, G., and Lipari, G., "Minimizing cpu energy in real-time systems with discrete speed management," *ACM Trans. Embed. Comput. Syst.*, vol. 8, pp. 31:1–31:23, July 2009.

[22] Chandrakasan, A., Sheng, S., and Brodersen, R., "Low-power cmos digital design," *Solid-State Circuits, IEEE Journal of*, vol. 27, pp. 473 –484, Apr. 1992.

[23] Chen, J.-J., "Expected energy consumption minimization in dvs systems with discrete frequencies," in *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, (New York, NY, USA), pp. 1720–1725, ACM, 2008.

[24] Chen, J.-J. and Thiele, L., "Expected system energy consumption minimization in leakage-aware dvs systems," in *Low Power Electronics and Design (ISLPED), 2008 ACM/IEEE International Symposium on*, pp. 315 –320, Aug. 2008.

[25] Corbet, J., "The cpuidle subsystem." LWN.net article, April 2010.

[26] Cucinotta, T., Palopoli, L., Abeni, L., Faggioli, D., and Lipari, G., "On the integration of application level and resource level qos control for real-time applications," *Industrial Informatics, IEEE Transactions on*, vol. 6, pp. 479 –491, Nov. 2010.

[27] Cucinotta, T., Palopoli, L., and Marzario, L., "Stochastic feedback-based control of qos in soft real-time systems," in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, vol. 4, pp. 3533–3538 Vol.4, Dec. 2004.

[28] Cucinotta, T., Palopoli, L., Marzario, L., Lipari, G., and Abeni, L., "Adaptive reservations in a linux environment," in *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pp. 238–245, may 2004.

[29] Dabiri, F., Vahdatpour, A., Potkonjak, M., and Sarrafzadeh, M., "Energy minimization for real-time systems with non-convex and discrete operation modes," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, (3001 Leuven, Belgium, Belgium), pp. 1416–1421, European Design and Automation Association, 2009.

[30] David, H., Fallin, C., Gorbatov, E., Hanebutte, U. R., and Mutlu, O., "Memory power management via dynamic voltage/frequency scaling," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, (New York, NY, USA), pp. 31–40, ACM, 2011.

[31] Devadas, V. and Aydin, H., "On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications," *Computers, IEEE Transactions on*, vol. 61, pp. 31 –44, Jan. 2012.

[32] Faggioli, D. and Checconi, F., "An edf scheduling class for the linux kernel," in *Proceedings of the Real-Time Linux Workshop*, September 2009.

[33] Fan, X., Ellis, C., and Lebeck, A., "Memory controller policies for dram power management," in *Proceedings of the 2001 international symposium on Low power electronics and design*, ISLPED '01, (New York, NY, USA), pp. 129–134, ACM, 2001.

[34] Freescale, *e500mc Core Reference Manual*, rev. 1 ed., Mar. 2012.

[35] Goel, A., Abeni, L., Krasic, C., Snow, J., and Walpole, J., "Supporting time-sensitive applications on a commodity os," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 165–180, dec. 2002.

[36] Greenhalgh, P., *Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7 (White Paper)*. ARM, October 2011.

[37] Gruian, F., "Hard real-time scheduling for low-energy using stochastic data and dvs processors," in *Proceedings of the 2001 international symposium on Low*

power electronics and design, ISLPED '01, (New York, NY, USA), pp. 46–51, ACM, 2001.

[38] Harris, R., Chabries, D., and Bishop, F., "A variable step (vs) adaptive filter algorithm," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 34, no. 2, pp. 309–316, 1986.

[39] Hayes, M. H., *Statistical Digital Signal Processing and Modeling*. John Wiley & Sons, Ltd., 2004.

[40] Hopper, J., "Reduce linux power consumption, part 1: The cpufreq subsystem," tech. rep., IBM, Tech. Rep.[Online]. Available: http://www. ibm. com/developerworks/linux/library/l-cpufreq-1/index. html, 2009.

[41] Intel, *Intel 64 and IA-32 Architectures Software Developers Manual Voume 3: System Programming Guide*, May 2012.

[42] Isci, C., Contreras, G., and Martonosi, M., "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 359–370, IEEE Computer Society, 2006.

[43] Jejurikar, R., Pereira, C., and K.Gupta, R., "Leakage aware dynamic voltage scaling for real time embedded systems," tech. rep., University of California at Irvine, Irvine, CA, USA, Nov. 2003.

[44] Khalilzad, N. M., Behnam, M., Nolte, T., and Åsberg, M., "On adaptive hierarchical scheduling of real-time systems using a feedback controller," in *Proceedings of the 3rd Workshop on Adaptive and Reconfigurable Embedded Systems*, APRES'11, Apr. 2011.

[45] Kim, N., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J., Irwin, M., Kandemir, M., and Narayanan, V., "Leakage current: Moore's law meets static power," *Computer*, vol. 36, pp. 68 – 75, Dec. 2003.

[46] Knuth, D. E., *The Art of Computer Programming, Seminumerical Algorithms*, vol. 2. Pearson Education, 1998.

[47] Lawitzky, M., Snowdon, D., and Petters, S., "Integrating real time and power management in a real system," in *Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 35–44, 2008.

[48] Lindberg, M., "A survey of reservation-based scheduling," Tech. Rep. ISRN LUTFD2/TFRT- -7618- -SE., Department of Automatic Control, Lund University, Sweden, Oct. 2007.

[49] Lipari, G. and Baruah, S., "Greedy reclamation of unused bandwidth constant-bandwidth servers," in *Proceedings of the 12th Euromicro conference on Real-time systems*, Euromicro-RTS'00, (Washington, DC, USA), pp. 193–200, IEEE Computer Society, 2000.

[50] Liu, C. L. and Layland, J. W., "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, Jan. 1973.

[51] McCord, J. R. and Moroney, R. M., *Introduction to Probability Theory*. The Macmillan Company, New York, 1965.

[52] McKenney, P., "A realtime preemption overview," August 2005.

[53] Mercer, C. W., Savage, S., and Tokuda, H., "Processor capacity reserves for multimedia operating systems," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1993.

[54] Moore, A., *Hybrid-SMP (White Paper)*. MARVELL, June 2012.

[55] Nascimento, V., "Improving the initial convergence of adaptive filters: variable-length lms algorithms," in *Digital Signal Processing, 2002. DSP 2002. 2002 14th International Conference on*, vol. 2, pp. 667–670 vol.2, 2002.

[56] Ngo, H. Q., "Tail and concentration inequalities." Lecture Notes on Probabilistic Analysis and Randomized Algorithms, SUNY at Buffalo, February 2011.

[57] Nguyen, H., Rivas, R., and Nahrstedt, K., "idsrt: Integrated dynamic soft real-time architecture for critical infrastructure data delivery over wlan," *Mobile Networks and Applications*, vol. 16, pp. 96–108, 2011.

[58] Pallipadi, V., Li, S., and Belay, A., "cpuidle: Do nothing, efficiently," in *Proceedings of the Linux Symposium*, vol. 2, 2007.

[59] Palopoli, L., Cucinotta, T., Marzario, L., and Lipari, G., "Aquosa adaptive quality of service architecture," *Software: Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[60] Pillai, P. and Shin, K. G., "Real-time dynamic voltage scaling for low-power embedded operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 89–102, Oct. 2001.

[61] Qian, D., Zhang, Z., and Hu, C., "Energy-aware task scheduling for real-time systems with discrete frequencies," *IEICE Transactions on Information and Systems*, vol. E94, Apr. 2011.

[62] Riera-Palou, F., Noras, J., and Cruickshank, D. G. M., "Linear equalisers, with dynamic and automatic length selection," *Electronics Letters*, vol. 37, no. 25, pp. 1553–1554, 2001.

[63] Roitzsch, M. and Pohlack, M., "Principles for the prediction of video decoding times applied to mpeg-1/2 and mpeg-4 part 2 video," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pp. 271–280, 2006.

[64] Snowdon, D., Ruocco, S., and Heiser, G., "Power management and dynamic voltage scaling: Myths and facts," in *2005 Workshop on Power Aware Real-time Computing*, 2005.

[65] Snowdon, D. C., Linden, G. V. D., Petters, S. M., and Heiser, G., "Accurate run-time prediction of performance degradation under frequency scaling," in *In Proceedings of the 2007 Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2007.

[66] Snowdon, D. C., Petters, S. M., and Heiser, G., "Accurate on-line prediction of processor and memoryenergy usage under voltage scaling," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, (New York, NY, USA), pp. 84–93, ACM, 2007.

[67] Song, I., Kim, S., and Karray, F., "A real-time scheduler design for a class of embedded systems," *Mechatronics, IEEE/ASME Transactions on*, vol. 13, pp. 36–45, Feb. 2008.

[68] Torres, G., "Everything you need to know about the cpu c-states power saving modes." Online Article, September 2008.

[69] Vardhan, V., Yuan, W., Harris, A., Adve, S., Kravets, R., Nahrstedt, K., Sachs, D., and Jones, D., "Grace-2: integrating fine-grained application adaptation with global adaptation for saving energy," *international Journal of embedded Systems*, vol. 4, no. 2, pp. 152–169, 2009.

[70] Xu, R., Mossé, D., and Melhem, R., "Minimizing expected energy consumption in real-time systems through dynamic voltage scaling," *ACM Trans. Comput. Syst.*, vol. 25, Dec. 2007.

[71] Yuan, W., Nahrstedt, K., Adve, S., Jones, D., and Kravets, R., "Grace-1: cross-layer adaptation for multimedia quality and battery energy," *Mobile Computing, IEEE Transactions on*, vol. 5, pp. 799 – 815, July 2006.

[72] Yuan, W. and Nahrstedt, K., "Energy-efficient soft real-time cpu scheduling for mobile multimedia systems," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 149–163, Oct. 2003.

[73] Yuan, W. and Nahrstedt, K., "Practical voltage scaling for mobile multimedia devices," in *Proceedings of the 12th annual ACM international conference on Multimedia*, MULTIMEDIA '04, (New York, NY, USA), pp. 924–931, ACM, 2004.

[74] Zamani, R. and Afsahi, A., "Adaptive estimation and prediction of power and performance inhigh performance computing," *Computer Science - Research and Development*, vol. 25, pp. 177–186, 2010.

[75] Zhu, Y. and Mueller, F., "Feedback edf scheduling of real-time tasks exploiting dynamic voltage scaling," *Real-Time Systems*, vol. 31, pp. 33–63, 2005. 10.1007/s11241-005-2744-3.